

2ND EDITION

WRITE GREAT CODE / VOLUME 1

UNDERSTANDING THE MACHINE

RANDALL HYDE



PRAISE FOR THE FIRST EDITION OF *WRITE GREAT CODE, VOLUME 1*

“Today’s programmers can hardly keep up with the race against inhumane deadlines and new technologies; therefore, they rarely have a chance to learn the basics of computer architectures and the inner workings of their programming languages. This book fills in the gaps. I strongly recommend it.”

—INFORMIT.COM

“[*Write Great Code*] isn’t your typical ‘teach yourself to program’ book. . . It’s relevant to all languages, and all levels of programming experience. . . Run, don’t walk, to buy and read this book.”

—BAY AREA LARGE INSTALLATION SYSTEM ADMINISTRATORS (BAYLISA)

5/5 stars: “[*Write Great Code*] fills in the blanks nicely and really could be part of a computer science degree required reading set. . . . Once this book is read, you will have a greater understanding and appreciation for code that is written efficiently—and you may just know enough to do that yourself. At least you will have a great start at the art of crafting efficient software.”

—MACCOMPANION

“Great fun to read.”

—VSJ MAGAZINE

“*Write Great Code, Volume 1: Understanding the Machine* should be on the required reading list for anyone who wants to develop terrific code in any language without having to learn assembly language.”

—WEBSERVERTALK

WRITE GREAT CODE

**VOLUME 1
2ND EDITION**

Understanding the Machine

by Randall Hyde



**no starch
press**

San Francisco

WRITE GREAT CODE, Volume 1: Understanding the Machine, 2nd Edition.

Copyright © 2020 by Randall Hyde.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-71850-036-X

ISBN-13: 978-1-71850-036-5

Publisher: William Pollock

Executive Editor: Barbara Yien

Production Editor: Rachel Monaghan

Developmental Editor: Athabasca Witschi

Project Editor: Dapinder Dosanjh

Cover and Interior Design: Octopod Studios

Technical Reviewer: Anthony Tribelli

Copyeditor: Rachel Monaghan

Compositor: Danielle Foster

Proofreader: James Fraleigh

Illustrator: David Van Ness

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1.415.863.9900; info@nostarch.com

www.nostarch.com

The Library of Congress issued the following Cataloging-in-Publication Data for the first edition of Volume 1:

Hyde, Randall.

Write great code : understanding the machine / Randall Hyde.

p. cm.

ISBN 1-59327-003-8

1. Computer programming. 2. Computer architecture. I. Title.

QA76.6.H94 2004

005.1--dc22

2003017502

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Randall Hyde is the author of *The Art of Assembly Language* and *Write Great Code, Volumes 1, 2, and 3* (all from No Starch Press), as well as *Using 6502 Assembly Language* and *P-Source* (Datamost). He is also the coauthor of *Microsoft Macro Assembler 6.0 Bible* (The Waite Group). Over the past 40 years, Hyde has worked as an embedded software/hardware engineer developing instrumentation for nuclear reactors, traffic control systems, and other consumer electronics devices. He has also taught computer science at California State Polytechnic University, Pomona, and at the University of California, Riverside. His website is www.randallhyde.com/.

About the Technical Reviewer

Tony Tribelli has more than 35 years of experience in software development, including work on embedded device kernels and molecular modeling. He developed video games for 10 years at Blizzard Entertainment. He is currently a software development consultant and privately develops applications utilizing computer vision.

BRIEF CONTENTS

Acknowledgments	xvii
Chapter 1: What You Need to Know to Write Great Code	1
Chapter 2: Numeric Representation	9
Chapter 3: Binary Arithmetic and Bit Operations	37
Chapter 4: Floating-Point Representation	61
Chapter 5: Character Representation	95
Chapter 6: Memory Organization and Access.	131
Chapter 7: Composite Data Types and Memory Objects.	159
Chapter 8: Boolean Logic and Digital Design	217
Chapter 9: CPU Architecture	251
Chapter 10: Instruction Set Architecture	283
Chapter 11: Memory Architecture and Organization	319
Chapter 12: Input and Output	349
Chapter 13: Computer Peripheral Buses	367
Chapter 14: Mass Storage Devices and Filesystems	381
Chapter 15: Miscellaneous Input and Output Devices.	413
Afterword: Thinking Low-Level, Writing High-Level	425
Appendix A: ASCII Character Set	427
Glossary	431
Index	443

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

XVII

1

WHAT YOU NEED TO KNOW TO WRITE GREAT CODE 1

1.1	The Write Great Code Series	1
1.2	What This Book Covers	2
1.3	Assumptions This Book Makes	4
1.4	Characteristics of Great Code	5
1.5	The Environment for This Book	6
1.6	Additional Tips	6
1.7	For More Information	7

2

NUMERIC REPRESENTATION 9

2.1	What Is a Number?	9
2.2	Numbering Systems	11
2.2.1	The Decimal Positional Numbering System	11
2.2.2	Radix (Base) Values	12
2.2.3	The Binary Numbering System	13
2.2.4	The Hexadecimal Numbering System	15
2.2.5	The Octal Numbering System	17
2.3	Numeric/String Conversions	18
2.4	Internal Numeric Representation	19
2.4.1	Bits	20
2.4.2	Bit Strings	20
2.5	Signed and Unsigned Numbers	23
2.6	Useful Properties of Binary Numbers	24
2.7	Sign Extension, Zero Extension, and Contraction	26
2.8	Saturation	28
2.9	Binary-Coded Decimal Representation	29
2.10	Fixed-Point Representation	30
2.11	Scaled Numeric Formats	33
2.12	Rational Representation	35
2.13	For More Information	35

3

BINARY ARITHMETIC AND BIT OPERATIONS 37

3.1	Arithmetic Operations on Binary and Hexadecimal Numbers	37
3.1.1	Adding Binary Values	38
3.1.2	Subtracting Binary Values	39
3.1.3	Multiplying Binary Values	40
3.1.4	Dividing Binary Values	41

3.2	Logical Operations on Bits	42
3.3	Logical Operations on Binary Numbers and Bit Strings	44
3.4	Useful Bit Operations	45
3.4.1	Testing Bits in a Bit String Using AND	45
3.4.2	Testing a Set of Bits for Zero/Not Zero Using AND	45
3.4.3	Comparing a Set of Bits Within a Binary String	46
3.4.4	Creating Modulo- <i>n</i> Counters Using AND	47
3.5	Shifts and Rotates	48
3.6	Bit Fields and Packed Data	51
3.7	Packing and Unpacking Data	55
3.8	For More Information	59

4

FLOATING-POINT REPRESENTATION

61

4.1	Introduction to Floating-Point Arithmetic	61
4.2	IEEE Floating-Point Formats	66
4.2.1	Single-Precision Floating-Point Format	67
4.2.2	Double-Precision Floating-Point Format	69
4.2.3	Extended-Precision Floating-Point Format	69
4.2.4	Quad-Precision Floating-Point Format	70
4.3	Normalization and Denormalized Values	70
4.4	Rounding	71
4.5	Special Floating-Point Values	73
4.6	Floating-Point Exceptions	74
4.7	Floating-Point Operations	75
4.7.1	Floating-Point Representation	75
4.7.2	Floating-Point Addition and Subtraction	75
4.7.3	Floating-Point Multiplication and Division	86
4.8	For More Information	93

5

CHARACTER REPRESENTATION

95

5.1	Character Data	96
5.1.1	The ASCII Character Set	96
5.1.2	The EBCDIC Character Set	99
5.1.3	Double-Byte Character Sets	100
5.1.4	The Unicode Character Set	101
5.1.5	Unicode Code Points	101
5.1.6	Unicode Code Planes	102
5.1.7	Surrogate Code Points	102
5.1.8	Glyphs, Characters, and Grapheme Clusters	103
5.1.9	Unicode Normals and Canonical Equivalence	105
5.1.10	Unicode Encodings	107
5.1.11	Unicode Combining Characters	108
5.2	Character Strings	110
5.2.1	Character String Formats	110
5.2.2	Types of Strings: Static, Pseudo-Dynamic, and Dynamic	116
5.2.3	Reference Counting for Strings	117
5.2.4	Delphi Strings	118
5.2.5	Custom String Formats	119

5.3	Character Set Data Types	119
5.3.1	Powerset Representation of Character Sets	119
5.3.2	List Representation of Character Sets	120
5.4	Designing Your Own Character Set	121
5.4.1	Designing an Efficient Character Set	122
5.4.2	Grouping the Character Codes for Numeric Digits	123
5.4.3	Grouping Alphabetic Characters	123
5.4.4	Comparing Alphabetic Characters	125
5.4.5	Grouping Other Characters	127
5.5	For More Information	130

6

MEMORY ORGANIZATION AND ACCESS

131

6.1	The Basic System Components.	131
6.1.1	The System Bus	132
6.2	Physical Organization of Memory	135
6.2.1	8-Bit Data Buses.	137
6.2.2	16-Bit Data Buses.	138
6.2.3	32-Bit Data Buses.	140
6.2.4	64-Bit Data Buses.	141
6.2.5	Small Accesses on Non-80x86 Processors	141
6.3	Big-Endian vs. Little-Endian Organization	142
6.4	The System Clock.	147
6.4.1	Memory Access and the System Clock	148
6.4.2	Wait States	150
6.4.3	Cache Memory	151
6.5	CPU Memory Access	154
6.5.1	The Direct Memory Addressing Mode	155
6.5.2	The Indirect Addressing Mode	155
6.5.3	The Indexed Addressing Mode	156
6.5.4	The Scaled-Index Addressing Modes	157
6.6	For More Information	157

7

COMPOSITE DATA TYPES AND MEMORY OBJECTS

159

7.1	Pointer Types.	159
7.1.1	Pointer Implementation	160
7.1.2	Pointers and Dynamic Memory Allocation	161
7.1.3	Pointer Operations and Pointer Arithmetic	162
7.2	Arrays	166
7.2.1	Array Declarations.	167
7.2.2	Array Representation in Memory	170
7.2.3	Accessing Elements of an Array	171
7.2.4	Multidimensional Arrays	172
7.3	Records/Structures	181
7.3.1	Records in Pascal/Delphi	181
7.3.2	Records in C/C++	182
7.3.3	Records in HLA	183
7.3.4	Records (Tuples) in Swift	183
7.3.5	Memory Storage of Records	184

7.4	Discriminant Unions	187
7.4.1	Unions in C/C++.	187
7.4.2	Unions in Pascal/Delphi	187
7.4.3	Unions in Swift	189
7.4.4	Unions in HLA	190
7.4.5	Memory Storage of Unions	190
7.4.6	Other Uses of Unions	191
7.5	Classes.	192
7.5.1	Inheritance	194
7.5.2	Class Constructors	198
7.5.3	Polymorphism	201
7.5.4	Abstract Methods and Abstract Base Classes.	202
7.6	Classes in C++	205
7.6.1	Abstract Member Functions and Classes in C++	206
7.6.2	Multiple Inheritance in C++.	206
7.7	Classes in Java	208
7.8	Classes in Swift	209
7.9	Protocols and Interfaces	210
7.10	Generics and Templates	213
7.11	For More Information	215

8

BOOLEAN LOGIC AND DIGITAL DESIGN

217

8.1	Boolean Algebra	218
8.1.1	The Boolean Operators.	218
8.1.2	Boolean Postulates	218
8.1.3	Boolean Operator Precedence.	220
8.2	Boolean Functions and Truth Tables	220
8.3	Function Numbers	222
8.4	Algebraic Manipulation of Boolean Expressions	223
8.5	Canonical Forms	224
8.5.1	Sum-of-Minterms Canonical Form and Truth Tables.	225
8.5.2	Algebraically Derived Sum-of-Minterms Canonical Form	227
8.5.3	Product-of-Maxterms Canonical Form	228
8.6	Simplification of Boolean Functions	229
8.7	What Does This Have to Do with Computers, Anyway?	237
8.7.1	Correspondence Between Electronic Circuits and Boolean Functions	238
8.7.2	Combinatorial Circuits	239
8.7.3	Sequential and Clocked Logic	245
8.8	For More Information	249

9

CPU ARCHITECTURE

251

9.1	Basic CPU Design	252
9.2	Decoding and Executing Instructions: Random Logic vs. Microcode	254
9.3	Executing Instructions, Step by Step	255
9.3.1	The mov Instruction.	256
9.3.2	The add Instruction.	257

9.3.3	The jnz Instruction	258
9.3.4	The loop Instruction	259
9.4	RISC vs. CISC: Improving Performance by Executing More, Faster, Instructions . .	259
9.5	Parallelism: The Key to Faster Processing	260
9.5.1	Functional Units	263
9.5.2	The Prefetch Queue	265
9.5.3	Conditions That Hinder the Performance of the Prefetch Queue	267
9.5.4	Pipelining: Overlapping the Execution of Multiple Instructions	267
9.5.5	Instruction Caches: Providing Multiple Paths to Memory	272
9.5.6	Pipeline Hazards	273
9.5.7	Superscalar Operation: Executing Instructions in Parallel.	275
9.5.8	Out-of-Order Execution	277
9.5.9	Register Renaming	277
9.5.10	VLIW Architecture	278
9.5.11	Parallel Processing	279
9.5.12	Multiprocessing	280
9.6	For More Information	281

10

INSTRUCTION SET ARCHITECTURE 283

10.1	The Importance of Instruction Set Design	284
10.2	Basic Instruction Design Goals.	285
10.2.1	Choosing Opcode Length	287
10.2.2	Planning for the Future	289
10.2.3	Choosing Instructions	289
10.2.4	Assigning Opcodes to Instructions	290
10.3	The Y86 Hypothetical Processor.	291
10.3.1	Y86 Limitations	291
10.3.2	Y86 Instructions	291
10.3.3	Operand Types and Addressing Modes on the Y86.	293
10.3.4	Encoding Y86 Instructions	293
10.3.5	Examples of Encoding Y86 Instructions.	296
10.3.6	Extending the Y86 Instruction Set.	300
10.4	Encoding 80x86 Instructions	301
10.4.1	Encoding Instruction Operands	303
10.4.2	Encoding the add Instruction	310
10.4.3	Encoding Immediate (Constant) Operands on the x86	314
10.4.4	Encoding 8-, 16-, and 32-Bit Operands	315
10.4.5	Encoding 64-Bit Operands	316
10.4.6	Alternate Encodings for Instructions	316
10.5	Implications of Instruction Set Design to the Programmer.	317
10.6	For More Information	317

11

MEMORY ARCHITECTURE AND ORGANIZATION 319

11.1	The Memory Hierarchy.	319
11.2	How the Memory Hierarchy Operates	322
11.3	Relative Performance of Memory Subsystems.	324

11.4	Cache Architecture	326
11.4.1	Direct-Mapped Cache	326
11.4.2	Fully Associative Cache	327
11.4.3	n-Way Set Associative Cache	328
11.4.4	Cache-Line Replacement Policies	329
11.4.5	Cache Write Policies	330
11.4.6	Cache Use and Software	331
11.5	NUMA and Peripheral Devices	332
11.6	Virtual Memory, Memory Protection, and Paging	332
11.7	Writing Software That Is Cognizant of the Memory Hierarchy	336
11.8	Runtime Memory Organization	338
11.8.1	Static and Dynamic Objects, Binding, and Lifetime	339
11.8.2	The Code, Read-Only, and Constant Sections	340
11.8.3	The Static Variables Section	341
11.8.4	The Storage Variables Section	341
11.8.5	The Stack Section	342
11.8.6	The Heap Section and Dynamic Memory Allocation	342
11.9	For More Information	348

12

INPUT AND OUTPUT

349

12.1	Connecting a CPU to the Outside World	350
12.2	Other Ways to Connect Ports to the System	353
12.3	I/O Mechanisms	354
12.3.1	Memory-Mapped I/O	354
12.3.2	I/O-Mapped Input/Output	355
12.3.3	Direct Memory Access	355
12.4	I/O Speed Hierarchy	356
12.5	System Buses and Data Transfer Rates	357
12.5.1	Performance of the PCI Bus	359
12.5.2	Performance of the ISA Bus	359
12.5.3	The AGP Bus	360
12.6	Buffering	360
12.7	Handshaking	361
12.8	Timeouts on an I/O Port	362
12.9	Interrupts and Polled I/O	363
12.10	Protected-Mode Operation and Device Drivers	364
12.10.1	The Device Driver Model	365
12.10.2	Communication with Device Drivers	365
12.11	For More Information	366

13

COMPUTER PERIPHERAL BUSES

367

13.1	The Small Computer System Interface	367
13.1.1	Limitations	368
13.1.2	Improvements	369
13.1.3	SCSI Protocol	370
13.1.4	SCSI Advantages	372

13.2	The IDE/ATA Interface	372
13.2.1	The SATA Interface	374
13.2.2	Fibre Channel	374
13.3	The Universal Serial Bus	374
13.3.1	USB Design.	374
13.3.2	USB Performance	376
13.3.3	Types of USB Transmissions.	377
13.3.4	USB-C	379
13.3.5	USB Device Drivers	380
13.4	For More Information	380

14

MASS STORAGE DEVICES AND FILESYSTEMS 381

14.1	Disk Drives	381
14.1.1	Floppy Disk Drives	382
14.1.2	Hard Drives.	382
14.1.3	RAID Systems	388
14.1.4	Optical Drives	389
14.1.5	CD, DVD, and Blu-ray Drives	390
14.2	Tape Drives.	392
14.3	Flash Storage	393
14.4	RAM Disks	395
14.5	Solid-State Drives	395
14.6	Hybrid Drives	396
14.7	Filesystems on Mass Storage Devices	396
14.7.1	Sequential Filesystems	397
14.7.2	Efficient File Allocation Strategies.	399
14.8	Writing Software That Manipulates Data on a Mass Storage Device	407
14.8.1	File Access Performance	407
14.8.2	Synchronous and Asynchronous I/O	409
14.8.3	The Implications of I/O Type.	409
14.8.4	Memory-Mapped Files	410
14.9	For More Information	411

15

MISCELLANEOUS INPUT AND OUTPUT DEVICES 413

15.1	Exploring Specific PC Peripheral Devices	413
15.1.1	The Keyboard	414
15.1.2	The Standard PC Parallel Port	415
15.1.3	Serial Ports	417
15.2	Mice, Trackpads, and Other Pointing Devices	417
15.3	Joysticks and Game Controllers	419
15.4	Sound Cards	419
15.4.1	How Audio Interface Peripherals Produce Sound	420
15.4.2	The Audio and MIDI File Formats	422
15.4.3	Programming Audio Devices	423
15.5	For More Information	424

AFTERWORD: THINKING LOW-LEVEL, WRITING HIGH-LEVEL	425
A	
ASCII CHARACTER SET	427
GLOSSARY	431
INDEX	443

ACKNOWLEDGMENTS

Many people have read and reread every word, symbol, and punctuation mark in this book in order to produce a better result. Kudos to the following people for their careful work on the second edition: development editor Athabasca Witschi, copyeditor/production editor Rachel Monaghan, and proofreader James Fraleigh.

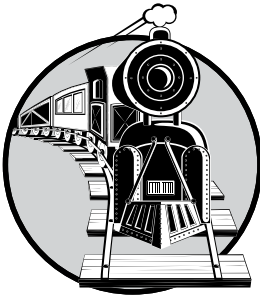
I would like to take the opportunity to graciously thank Anthony Tribelli, a longtime friend, who went well beyond the call of duty when doing a technical review of this book. He pulled every line of code out of this book (including snippets) and compiled and ran it to make sure it worked properly. His suggestions and opinions throughout the technical review process have dramatically improved the quality of this work.

Of course, I would also like to thank all the countless readers over the years who've emailed suggestions and corrections, many of which have found their way into this second edition.

Thanks to all of you,
Randall Hyde

1

WHAT YOU NEED TO KNOW TO WRITE GREAT CODE



The *Write Great Code (WGC)* series will teach you how to write code you can be proud of; code that will impress other programmers, satisfy customers, and prove popular with users; and code that people (customers, your boss, and so on) won't mind paying top dollar to obtain.

In general, the books in the *WGC* series will discuss how to write software that achieves legendary status, eliciting the awe and admiration of other programmers.

1.1 The Write Great Code Series

Write Great Code, Volume 1: Understanding the Machine (WGC1 hereafter) is the first of six books in the *WGC* series. Writing great code requires a combination of knowledge, experience, and skill that programmers usually obtain only after years of mistakes and discoveries. The purpose of this series is to share with both new and experienced programmers a few decades' worth of

observations and experience. I hope that these books will help reduce the time and frustration it takes to learn things “the hard way.”

This book, *WGC1*, fills in the low-level details that are often skimmed over in a typical computer science or engineering curriculum. These details are the foundation for the solutions to many problems, and you can’t write efficient code without this information. Though I’m attempting to keep each book independent, *WGC1* might be considered a prerequisite for the subsequent volumes in the series.

Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level (WGC2) immediately applies the knowledge from this book. *WGC2* will teach you how to analyze code written in a high-level language to determine the quality of the machine code that a compiler would generate for it. Optimizing compilers don’t always generate the best machine code possible—the statements and data structures you choose in your source files can have a big impact on the efficiency of the compiler’s output. *WGC2* will teach you how to write efficient code without resorting to assembly language.

There are many attributes of great code besides efficiency, and the third book in this series, *Write Great Code, Volume 3: Engineering Software (WGC3)*, will cover some of those. *WGC3* will discuss software development metaphors, development methodologies, types of developers, system documentation, and the Unified Modeling Language (UML). *WGC3* provides the basis for personal software engineering.

Great code begins with a great design. *Write Great Code, Volume 4: Designing Great Code (WGC4)*, will describe the process of analysis and design (both structured and object-oriented). *WGC4* will teach you how to translate an initial concept into a working design for your software systems.

Write Great Code, Volume 5: Great Coding (WGC5) will teach you how to create source code that others can easily read and maintain, as well as how to improve your productivity without the burden of the “busy work” that many software engineering books discuss.

Great code *works*. Therefore, I’d be remiss not to include a book on testing, debugging, and quality assurance. Few programmers properly test their code. This generally isn’t because they find testing boring or beneath them, but because they don’t know *how* to test their programs, eradicate defects, and ensure the quality of their code. To help overcome this problem, *Write Great Code, Volume 6: Testing, Debugging, and Quality Assurance (WGC6)* will describe how to efficiently test your applications without all the drudgery engineers normally associate with this task.

1.2 What This Book Covers

In order to write great code, you need to know how to write efficient code, and to write efficient code, you must understand how computer systems execute programs and how abstractions in programming languages map to the low-level hardware capabilities of the machine.

In the past, learning great coding techniques has required learning assembly language. While this isn’t a bad approach, it’s overkill. Learning

assembly language involves learning two related subjects: machine organization, and programming in assembly language. The real benefits of learning assembly language come from the machine organization component. Thus, this book focuses solely on machine organization so you can learn to write great code without the overhead of also learning assembly language.

Machine organization is a subset of computer architecture that covers low-level data types, internal CPU organization, memory organization and access, low-level machine operations, mass storage organization, peripherals, and how computers communicate with the rest of the world. This book concentrates on those parts of computer architecture and machine organization that are visible to the programmer or are helpful for understanding why system architects chose a particular system design. The goal of learning machine organization, and of this book, is not to enable you to design your own CPU or computer system, but to equip you to make the most efficient use of existing computer designs. Let's do a quick run-through of the specific topics we'll cover.

Chapters 2, 4, and 5 deal with basic computer data representation—how computers represent signed and unsigned integer values, characters, strings, character sets, real values, fractional values, and other numeric and non-numeric quantities. Without a solid grasp of how computers represent these various data types internally, it'll be difficult for you to understand why some operations that use these data types are so inefficient.

Chapter 3 discusses binary arithmetic and bit operations used by most modern computer systems. It also offers several insights into how you can write better code by using arithmetic and logical operations in ways not normally taught in beginning programming courses. Learning these kinds of standard “tricks” is part of how you become a great programmer.

Chapter 6 introduces memory, discussing how the computer accesses its memory and describing characteristics of memory performance. This chapter also covers various machine code *addressing modes*, which CPUs use to access different types of data structures in memory. In modern applications, poor performance often occurs because the programmer, unaware of the ramifications of memory access in their programs, creates bottlenecks. Chapter 6 addresses many of these ramifications.

Chapter 7 returns to data types and representation by covering composite data types and memory objects: pointers, arrays, records, structures, and unions. All too often, programmers use large composite data structures without even considering the memory and performance impact of doing so. The low-level description of these high-level composite data types will make clear their inherent costs, so you can use them sparingly and wisely.

Chapter 8 discusses Boolean logic and digital design. This chapter provides the mathematical and logical background you'll need to understand the design of CPUs and other computer system components. In particular, this chapter discusses how to optimize Boolean expressions, such as those found in common high-level programming language statements like `if` and `while`.

Continuing the hardware discussion from Chapter 8, Chapter 9 discusses CPU architecture. A basic understanding of CPU design and operation is essential if you want to write great code. By writing your code in a manner consistent with how a CPU will execute it, you'll get much better performance using fewer system resources.

Chapter 10 discusses CPU instruction set architecture. Machine instructions are the primitive units of execution on any CPU, and the duration of program execution is directly determined by the number and type of machine instructions the CPU must process. Learning how computer architects design machine instructions can provide valuable insight into why certain operations take longer to execute than others. Once you understand the limitations of machine instructions and how the CPU interprets them, you can use this information to turn mediocre code sequences into great ones.

Chapter 11 returns to the subject of memory, covering memory architecture and organization. This chapter is especially important for anyone wanting to write fast code. It describes the memory hierarchy and how to maximize the use of the cache and other fast memory components. You'll learn about thrashing and how to avoid low-performance memory access in your applications.

Chapters 12 through 15 describe how computer systems communicate with the outside world. Many peripheral (input/output) devices operate at much lower speeds than the CPU and memory. You could write the fastest-executing sequence of instructions possible, and your application would still run slowly because you didn't understand the limitations of the I/O devices in your system. These four chapters discuss generic I/O ports, system buses, buffering, handshaking, polling, and interrupts. They also explain how to efficiently use many popular PC peripheral devices, including keyboards, parallel (printer) ports, serial ports, disk drives, tape drives, flash storage, SCSI, IDE/ATA, USB, and sound cards.

1.3 Assumptions This Book Makes

This book was written with certain assumptions about your prior knowledge. You'll reap the greatest benefit from this material if your skill set matches the following:

- You should be reasonably competent in at least one modern programming language. This includes C/C++, C#, Java, Swift, Python, Pascal/Delphi (Object Pascal), BASIC, and assembly, as well as languages like Ada, Modula-2, and FORTRAN.
- Given a small problem description, you should be capable of working through the design and implementation of a software solution for that problem. A typical semester or quarter course at a college or university (or several months' experience on your own) should be sufficient background for this book.

At the same time, this book is not language specific; its concepts transcend whatever programming language(s) you're using. Furthermore, this book does not assume that you use or know any particular language. To help make the examples more accessible, the programming examples rotate among several languages. This book explains exactly how the example code operates so that even if you're unfamiliar with the specific programming language, you'll be able to understand its operation by reading the accompanying description.

This book uses the following languages and compilers in various examples:

- C/C++: GCC, Microsoft's Visual C++
- Pascal: Embarcadero's Delphi, Free Pascal
- Assembly language: Microsoft's MASM, HLA (High-Level Assembly), Gas (the Gnu Assembler; on the PowerPC and ARM)
- Swift 5 (Apple)
- Java (v6 or later)
- BASIC: Microsoft's Visual Basic

Often, the examples appear in multiple languages, so it's usually safe to ignore a specific example if you don't understand the syntax of the language it uses.

1.4 Characteristics of Great Code

Different programmers will have different definitions for great code, so it's impossible to provide an all-encompassing definition that will satisfy everyone. However, nearly everyone will agree that great code:

- Uses the CPU efficiently (that is, it's fast)
- Uses memory efficiently (that is, it's small)
- Uses system resources efficiently
- Is easy to read and maintain
- Follows a consistent set of style guidelines
- Uses an explicit design that follows established software engineering conventions
- Is easy to enhance
- Is well tested and robust (that is, it works)
- Is well documented

We could easily add dozens of items to this list. Some programmers, for example, may feel that great code must be portable, must follow a given set of programming style guidelines, or must be written in a certain language (or *not* be written in a certain language). Some may feel that great code must be written as simply as possible, while others believe that it must be

written quickly. Still others may feel that great code is created on time and under budget.

Here is the definition this book uses:

Great code is software that is written using a consistent and prioritized set of good software characteristics. In particular, great code follows a set of rules that guide the decisions a programmer makes when implementing an algorithm as source code.

Two different programs do not have to follow the same set of rules (that is, they need not possess the same set of characteristics) in order for both to be great. In one environment, the priority might be producing code that's portable across different CPUs and operating systems. In a different environment, efficiency (speed) might be the primary goal, and portability might not be an issue. Neither program would qualify as great according to the rules of the other, but as long as the software consistently follows the guidelines established for that particular program, you can argue that it is an example of great code.

1.5 The Environment for This Book

Although this book presents generic information, parts of the discussion will necessarily be specific to a particular system. Because the Intel Architecture PCs are, by far, the most common in use today, this book will use that platform when discussing specific system-dependent concepts.

Most of the specific examples in this book run on a late-model Intel Architecture (including AMD) CPU under macOS, Windows, or Linux, with a reasonable amount of RAM and other system peripherals normally found on a late-model PC. This book attempts to stick with standard library interfaces to the operating system (OS) wherever possible, and it makes OS-specific calls only when the alternative is to write “less than great” code. The concepts, if not the software itself, will apply to Android, Chrome, iOS, Macs, Unix boxes, embedded systems, and even mainframes, though you may need to research how to apply a concept to your platform.

1.6 Additional Tips

No single book can completely cover everything you need to know in order to write great code. This book, therefore, concentrates on the areas that are most pertinent for machine organization, providing the 90 percent solution for those who are interested in writing the best possible code. To get that last 10 percent you'll need additional help. Here are some suggestions:

Learn assembly language. Fluency in at least one assembly language will fill in many missing details that you just won't get by learning machine organization alone. Unless you plan to use assembly language in your software systems, you don't have to learn it on the platform(s) to which you're targeting your software. Probably your best bet is to

learn 80x86 assembly language on a PC, because there are lots of great software tools for learning Intel Architecture assembly language (for example, HLA) that simply don't exist on other platforms. The point of learning assembly language here is not to write assembly code, but to learn the assembly paradigm. If you know 80x86 assembly language, you'll have a good idea of how other CPUs (such as the ARM or the IA-64 family) operate.

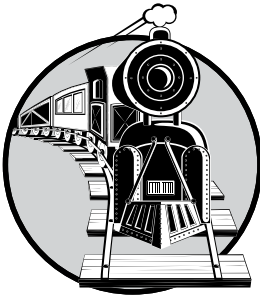
Study advanced computer architecture. Machine organization is a subset of computer architecture, but space limitations prevent full coverage of both in this book. While you may not need to know how to design your own CPUs, studying computer architecture might teach you something omitted here.

1.7 For More Information

- Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Morgan Kaufmann, 2012.
- Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

2

NUMERIC REPRESENTATION



High-level languages shield programmers from the pain of dealing with low-level numeric representation. Writing great code, however, requires that you understand how computers represent numbers, so that is the focus of this chapter. Once you understand internal numeric representation, you'll discover efficient ways to implement many algorithms and avoid the pitfalls associated with common programming practices.

2.1 What Is a Number?

Having taught assembly language programming for many years, I've discovered that most people don't understand the fundamental difference between a number and the representation of that number. Most of the time, this confusion is harmless. However, many algorithms depend on

the internal and external representations we use for numbers to operate correctly and efficiently. If you don't understand the difference between the abstract concept of a number and the representation of that number, you'll have trouble understanding, using, or creating such algorithms.

A *number* is an intangible, abstract concept. It is an intellectual device that we use to denote quantity. Let's say I told you that a book has one hundred pages. You could touch the pages—they are tangible. You could even count those pages to verify that there are one hundred of them. However, “one hundred” is simply an abstraction I'm applying to the book as a way of describing its size.

The important thing to realize is that the following is *not* one hundred:

100

This is nothing more than ink on paper forming certain lines and curves (called *glyphs*). You might recognize this sequence of symbols as a representation of one hundred, but this is not the actual value 100. It's just three symbols on this page. It isn't even the only representation for one hundred—consider the following, which are all different representations of the value 100:

100	Decimal representation
C	Roman numeral representation
64 ₁₆	Base-16 (hexadecimal) representation
1100100 ₂	Base-2 (binary) representation
144 ₈	Base-8 (octal) representation
one hundred	English representation

The representation of a number is (generally) some sequence of symbols. For example, the common representation of the value one hundred, “100,” is really a sequence of three numeric digits: the digit 1 followed by the digit 0 followed by a second 0 digit. Each of these digits has some specific meaning, but we could have just as easily used the sequence “64” to represent one hundred. Even the individual digits that make up this representation of 100 are not numbers. They are numeric digits, tools we use to represent numbers, but they are not numbers themselves.

Now you might be wondering why you should even care whether a sequence of symbols like “100” is the actual value one hundred or just the representation of it. The reason is that you'll encounter several different sequences of symbols in a computer program that look like numbers (that is, they look like “100”), and you don't want to confuse them with actual numeric values. Conversely, there are many different representations for the value one hundred that a computer could use, and it's important for you to realize that they are equivalent.

2.2 Numbering Systems

A *numbering system* is a mechanism we use to represent numeric values. Today, most people use the *decimal* (or *base-10*) numbering system, and most computer systems use the *binary* (or *base-2*) numbering system. Confusion between the two can lead to poor coding practices.

The Arabs developed the decimal numbering system we commonly use today (this is why the 10 decimal digits are known as *Arabic numerals*). The decimal system uses *positional notation* to represent values with a small group of different symbols. Positional notation gives meaning not only to the symbol itself, but also to the position of the symbol in the sequence of symbols—a scheme that is far superior to other, nonpositional, representations. To appreciate the difference between a positional system and a nonpositional system, consider the *tally-slash* representation of the number 25 in Figure 2-1.

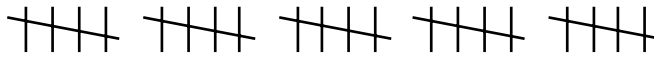
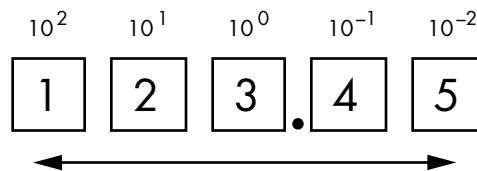


Figure 2-1: Tally-slash representation of 25

The tally-slash representation uses a sequence of n marks to represent the value n . To make the values easier to read, most people arrange the tally marks in groups of five, as in Figure 2-1. The advantage of the tally-slash numbering system is that it's easy to use for counting objects. However, the notation is bulky, and arithmetic operations are difficult. The biggest problem with the tally-slash representation is the amount of physical space it consumes. To represent the value n requires an amount of space proportional to n . Therefore, for large values of n , this notation becomes unusable.

2.2.1 The Decimal Positional Numbering System

The decimal positional numbering system represents numbers using strings of Arabic numerals, optionally including a decimal point to separate whole and fractional portions of the number representation. The position of a digit in the string affects its meaning: each digit to the left of the decimal point represents a value between 0 and 9, multiplied by an increasing power of 10 (see Figure 2-2). The symbol immediately to the left of the decimal point in the sequence represents a value between 0 and 9. If there are at least two digits, the second symbol to the left of the decimal point represents a value between 0 and 9 times 10, and so forth. To the right of the decimal point, the values decrease.



The magnitude associated with each digit is relative to its distance from the decimal point.

Figure 2-2: A positional numbering system

The numeric sequence 123.45 represents:

$$(1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (4 \times 10^{-1}) + (5 \times 10^{-2})$$

or:

$$100 + 20 + 3 + 0.4 + 0.05$$

To understand the power of the base-10 positional numbering system, consider that, compared to the tally-slash system:

- It can represent the value 10 in one-third the space.
- It can represent the value 100 in about 3 percent of the space.
- It can represent the value 1,000 in about 0.3 percent of the space.

As the numbers grow larger, the disparity becomes even greater. Because of their compact and easy-to-recognize notation, positional numbering systems are quite popular.

2.2.2 Radix (Base) Values

Humans developed the decimal numbering system because it corresponds to the number of fingers (“digits”) on their hands. However, decimal isn’t the only positional numbering system possible; in fact, for most computer-based applications, it isn’t even the best numbering system available. So, let’s take a look at how to represent values in other numbering systems.

The decimal positional numbering system uses powers of 10 and 10 unique symbols for each digit position. Because decimal numbers use powers of 10, we call them “base-10” numbers. By substituting a different set of numeric digits and multiplying those digits by powers of some base other than 10, we can devise a different numbering system. The base, or *radix*, is the value that we raise to successive powers for each digit to the left of the *radix point* (note that the term *decimal point* applies only to decimal numbers).

As an example, we can create a base-8 (*octal*) numbering system using eight symbols (0–7) and successive powers of 8. Consider the octal number 123₈ (the subscript denotes the base using standard mathematical notation), which is equivalent to 83₁₀:

$$1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0$$

or:

$$64 + 16 + 3$$

To create a base-*n* numbering system, you need *n* unique digits. The smallest possible radix is 2 (for this scheme). For bases 2 through 10, the convention is to use the Arabic digits 0 through *n* – 1 (for a base-*n* system). For bases greater than 10, the convention is to use the alphabetic digits *a* through *z* or *A* through *Z* (ignoring case) for digits greater than 9. This scheme supports numbering systems through base 36 (10 numeric digits and 26 alphabetic digits). There’s no agreed-upon convention for symbols beyond the 10 Arabic numeric digits and the 26 alphabetic digits.

Throughout this book, we'll deal with base-2, base-8, and base-16 values because base 2 (binary) is the native representation most computers use, base 8 was popular on older computer systems, and base 16 is more compact than base 2. You'll find that many programs use these three bases, so it's important to be familiar with them.

2.2.3 The Binary Numbering System

Since you're reading this book, chances are pretty good that you're already familiar with the base-2, or binary, numbering system; nevertheless, a quick review is in order. The binary numbering system works just like the decimal numbering system, except binary uses only the digits 0 and 1 (rather than 0–9) and uses powers of 2 (rather than powers of 10).

Why even worry about binary? After all, almost every computer language available allows programmers to use decimal notation (automatically converting decimal representation to the internal binary representation). Despite this capability, most modern computer systems talk to I/O devices using binary, and their arithmetic circuitry operates on binary data. Many algorithms depend upon binary representation for correct operation. In order to write great code, then, you'll need a complete understanding of binary representation.

2.2.3.1 Converting Between Decimal and Binary Representation

To appreciate what the computer does for you, it's useful to learn how to convert between decimal and binary representations manually.

To convert a binary value to decimal, add 2^i for each 1 in the binary string, where i is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents:

$$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

or:

$$128 + 64 + 8 + 2$$

or:

$$202_{10}$$

Converting decimal to binary is almost as easy. Here's an algorithm that converts decimal representation to the corresponding binary representation:

1. If the number is even, emit a 0. If the number is odd, emit a 1.
2. Divide the number by 2 and discard any fractional component or remainder.
3. If the quotient is 0, the algorithm is complete.
4. If the quotient is not 0 and the number is odd, insert a 1 before the current string. If the quotient is not 0 and the number is even, prefix your binary string with 0.
5. Go back to step 2 and repeat.

This example converts 202 to binary:

1. 202 is even, so emit a 0 and divide by 2 (101): 0
2. 101 is odd, so emit a 1 and divide by 2 (50): 10
3. 50 is even, so emit a 0 and divide by 2 (25): 010
4. 25 is odd, so emit a 1 and divide by 2 (12): 1010
5. 12 is even, so emit a 0 and divide by 2 (6): 01010
6. 6 is even, so emit a 0 and divide by 2 (3): 001010
7. 3 is odd, so emit a 1 and divide by 2 (1): 1001010
8. 1 is odd, so emit a 1 and divide by 2 (0): 11001010
9. The result is 0, so the algorithm is complete, producing 11001010.

2.2.3.2 Making Binary Numbers Easier to Read

As you can tell by the equivalent representations 202_{10} and 11001010_2 , binary representation is not as compact as decimal representation. We need some way to make the digits, or *bits*, in binary numbers less bulky and easier to read.

In the United States, most people separate every three digits with a comma to make larger numbers easier to read. For example, 1,023,435,208 is much easier to read and comprehend than 1023435208. This book will adopt a similar convention for binary numbers; each group of 4 binary bits will be separated with an underscore. For example, the binary value 101011110110010_2 will be written as `1010_1111_1011_00102`.

2.2.3.3 Representing Binary Values in Programming Languages

Thus far, this chapter has used the subscript notation embraced by mathematicians to denote binary values (the lack of a subscript indicates the decimal base). Subscripts are not generally recognized by program text editors or programming language compilers, however, so we need some other way to represent various bases within a standard ASCII text file.

Generally, only assembly language compilers (“assemblers”) allow the use of literal binary constants in a program.¹ Because assemblers vary widely, there are many different ways to represent binary literal constants in an assembly language program. This book presents examples using MASM and HLA, so it makes sense to adopt their conventions.

MASM represents binary values as a sequence of binary digits (0 and 1) ending with a `b` or `B`. The binary representation for 9 would be `1001b` in a MASM source file.

HLA prefixes binary values with the percent symbol (%). To make binary numbers more readable, HLA also allows you to insert underscores within binary strings like so:

```
%11_1011_0010_1101
```

1. Swift also allows you to specify binary numbers, using a `0b` prefix.

2.2.4 The Hexadecimal Numbering System

As noted earlier, binary number representation is verbose. Hexadecimal representation offers two great features: it's very compact, and it's easy to convert between binary and hexadecimal. Therefore, software engineers generally use hexadecimal representation rather than binary to make their programs more readable.

Because hexadecimal representation is base 16, each digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234_{16} is equal to:

$$1 \times 16^3 + 2 \times 16^2 + 3 \times 16^1 + 4 \times 16^0$$

or:

$$4096 + 512 + 48 + 4$$

or:

$$4660_{10}$$

Hexadecimal representation uses the letters *A* through *F* for the additional six digits it requires (above the 10 standard decimal digits, 0–9). The following are all examples of valid hexadecimal numbers:

234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

2.2.4.1 Representing Hexadecimal Values in Programming Languages

One problem with hexadecimal representation is that it's difficult to differentiate hexadecimal values like “DEAD” from standard program identifiers. Therefore, most programming languages use a special prefix or suffix character to denote hexadecimal values. Here's how you specify literal hexadecimal constants in several popular languages:

- The C, C++, C#, Java, Swift, and other C-derivative programming languages use the prefix `0x`. You'd use the character sequence `0xdead` for the hexadecimal value $DEAD_{16}$.
- The MASM assembler uses an `h` or `H` suffix. Because this doesn't completely resolve the ambiguity between certain identifiers and literal hexadecimal constants (for example, “`death`” still looks like an identifier to MASM), it also requires that a hexadecimal value begin with a numeric digit. So, you would add `0` to the beginning of the value (because a prefix of `0` does not alter the value of a numeric representation) to get `0death`, which unambiguously represents $DEAD_{16}$.
- Visual Basic uses the `&H` or `&h` prefix. Continuing with the current example, you'd use `&Hdead` to represent $DEAD_{16}$ in Visual Basic.
- Pascal (Delphi) uses the prefix `$`. So, you'd use `$dead` to represent the current example in Delphi/Free Pascal.
- HLA also uses the prefix `$`. As with binary numbers, it also allows you to insert underscores into a hexadecimal number to make it easier to read (for example, `$FDEC_A012`).

In general, this book will use the HLA/Delphi/Free Pascal format except in examples specific to other programming languages. Because there are several C/C++ examples in this book, you'll frequently see the C/C++ notation as well.

2.2.4.2 Converting Between Hexadecimal and Binary Representations

Another reason hexadecimal notation is popular is because it's easy to convert between the binary and hexadecimal representations. By memorizing the few simple rules shown in Table 2-1, you can mentally perform this conversion.

Table 2-1: Binary/Hexadecimal Conversion Chart

Binary	Hexadecimal
%0000	\$0
%0001	\$1
%0010	\$2
%0011	\$3
%0100	\$4
%0101	\$5
%0110	\$6
%0111	\$7
%1000	\$8
%1001	\$9
%1010	\$A
%1011	\$B
%1100	\$C
%1101	\$D
%1110	\$E
%1111	\$F

To convert the hexadecimal representation of a number into binary, substitute the corresponding 4 bits for each hexadecimal digit. For example, to convert \$ABCD into the binary form %1010_1011_1100_1101, convert each hexadecimal digit according to the values in Table 2-1:

A	B	C	D	Hexadecimal
1010	1011	1100	1101	Binary

Converting the binary representation of a number into hexadecimal is almost as easy. First, pad the binary number with 0s to make sure it is a multiple of 4 bits long. For example, given the binary number 1011001010, add two 0 bits to the left of the number to make it 12 bits without changing its value: 001011001010. Next, separate the binary value into groups of 4 bits: 0010_1100_1010. Finally, look up these binary values in Table 2-1 and

substitute the appropriate hexadecimal digits: \$2CA. As you can see, this is much simpler than converting between decimal and binary or between decimal and hexadecimal.

2.2.5 The Octal Numbering System

Octal (base-8) representation was common in early computer systems, so you might still see it in use now and then. Octal is great for 12-bit and 36-bit computer systems (or any other size that is a multiple of 3), but not particularly for computer systems whose bit size is a power of 2 (8-, 16-, 32-, and 64-bit computer systems). Nevertheless, some programming languages allow you to specify numeric values in octal notation, and you can still find some older Unix applications that use it.

2.2.5.1 Representing Octal Values in Programming Languages

The C programming language (and derivatives like C++ and Java), MASM, Swift, and Visual Basic support octal representation. You should be aware of the notation they use for octal numbers in case you come across it in programs written in these languages.

- In C, you specify the octal base by prefixing a numeric string with a 0 (zero). For example, 0123 is equivalent to the decimal value 83₁₀ and definitely *not* equivalent to the decimal value 123₁₀.
- MASM uses a Q or q suffix. (Microsoft/Intel probably chose *Q* because it looks like the letter *O* but isn't likely to be confused with a zero.)
- Swift uses a 0o prefix. For example, 0o14 represents the decimal value 12₁₀.
- Visual Basic uses the prefix &O (that's the letter *O*, not a zero). For example, you'd use &O123 to represent the decimal value 83₁₀.

2.2.5.2 Converting Between Octal and Binary Representation

Converting between binary and octal is similar to converting between binary and hexadecimal, except that you work in groups of 3 bits rather than 4. See Table 2-2 for the list of binary and octal equivalent representations.

Table 2-2: Binary/Octal Conversion Chart

Binary	Octal
%000	0
%001	1
%010	2
%011	3
%100	4
%101	5
%110	6
%111	7

To convert octal into binary, replace each octal digit in the number with the corresponding 3 bits from Table 2-2. For example, when you convert 123q into a binary value, the final result is %0_0101_0011:

1	2	3
001	010	011

To convert a binary number into octal, you break up the binary string into groups of 3 bits (padding with 0s, as necessary) and then replace each triad with the corresponding octal digit from Table 2-2.

To convert an octal value to hexadecimal notation, convert the octal number to binary and then convert the binary value to hexadecimal.

2.3 Numeric/String Conversions

In this section, we'll explore conversions from string to numeric form and vice versa. Because most programming languages (or their libraries) perform these conversions automatically, beginning programmers are often unaware that they're even taking place. For example, consider how easy it is to convert a string to numeric form in various languages:

```
cin >> i;                // C++
readln( i );              // Pascal
let j = Int(readLine() ?? "")! // Swift
input i                   // BASIC
stdin.get( i );           // HLA
```

In each of these statements, the variable `i` can hold some integer number. The input from the user's console, however, is a string of characters. The programming language's runtime library is responsible for converting that string of characters to the internal binary form the CPU requires. Note that Swift only allows you to read a string from the standard input; you must explicitly convert that string to an integer using the `Int()` constructor/type conversion function.

Unfortunately, if you have no idea of the cost of these statements, you won't realize how they can impact your program when performance is critical. It's important to understand the underlying work involved in the conversion algorithms so you won't frivolously use statements like these.

NOTE

For simplicity's sake, we'll discuss unsigned integer values and ignore the possibility of illegal characters and numeric overflow. Therefore, the following algorithms slightly understate the actual work involved.

Use this algorithm to convert a string of decimal digits to an integer value:

1. Initialize a variable with 0; this will hold the final value.
2. If there are no more digits in the string, then the algorithm is complete, and the variable holds the numeric value.

3. Fetch the next digit (moving from left to right) from the string and convert it from ASCII to an integer.
4. Multiply the variable by 10, and then add in the digit fetched in step 3.
5. Return to step 2 and repeat.

Converting an integer value to a string of characters takes even more effort:

1. Initialize a string to the empty string.
2. If the integer value is 0, output a 0, and the algorithm is complete.
3. Divide the current integer value by 10, computing the remainder and quotient.
4. Convert the remainder (always in the range $0..9^2$) to a character, and insert the character at the beginning of the string.
5. If the quotient is not 0, make it the new value and repeat steps 3–5.
6. Output the characters in the string.

The particulars of these algorithms are not important. What *is* important is that these steps execute once for each output character and division is very slow. So, a simple statement like one of the following can hide a fair amount of work from the programmer:

```
printf( "%d", i );    // C
cout << i;           // C++
print i              // BASIC
write( i );          // Pascal
print( i )           // Swift
stdout.put( i );     // HLA
```

To write great code, you don't need to avoid using numeric/string conversions altogether; however, a great programmer will take care to use them only as necessary.

Remember that these algorithms are valid only for unsigned integers. Signed integers require a little more effort to process (though the extra work is almost negligible). Floating-point values, however, are far more difficult to convert between string and numeric form, so keep that in mind when writing code that uses floating-point arithmetic.

2.4 Internal Numeric Representation

Most modern computer systems use an internal binary format to represent values and other objects. However, most systems can only efficiently represent binary values of a given size. In order to write great code, you need to make sure that your programs use data objects that the machine

2. The “..” notation, taken from Pascal and other programming languages, denotes a range of values. Thus, “0..9” denotes all integer values between 0 and 9.

can represent efficiently. This section will describe how computers physically represent values so you can design your programs accordingly.

2.4.1 Bits

The smallest unit of data on a binary computer is a single bit. Because a bit can represent only two different values (typically 0 or 1), you might assume that you can't use it for much. But in fact, there's an infinite number of two-item combinations you can represent with a single bit. Here are some examples (with arbitrary binary encodings I've created):

- Zero (0) or one (1)
- False (0) or true (1)
- Off (0) or on (1)
- Male (0) or female (1)
- Wrong (0) or right (1)

You're not limited to representing binary data types, either (that is, those objects that have only two distinct values). You could also use a single bit to represent any two distinct items:

- The numbers 723 (0) and 1,245 (1)
- The colors red (0) and blue (1)

You could even represent two unrelated objects with a single bit. For example, you could use the bit value 0 to represent the color red and the bit value 1 to represent the number 3,256. You can represent *any* two different values with a single bit—but *only* two different values. Therefore, individual bits aren't sufficient for most computational needs. To overcome the limitations of a single bit, we create *bit strings* from a sequence of multiple bits.

2.4.2 Bit Strings

By combining bits into a sequence, we can form binary representations that are equivalent to other representations of numbers, like hexadecimal and octal. Most computer systems don't let you combine an arbitrary number of bits, so you have to work with bit strings of certain fixed lengths.

A *nibble* is a collection of 4 bits. Most computer systems don't provide efficient access to nibbles in memory. Notably, it takes exactly 1 nibble to represent a single hexadecimal digit.

A *byte* is 8 bits and is the smallest addressable data item on many CPUs; that is, the CPU can efficiently retrieve data in groups of 8 bits from memory. For this reason, the smallest data type that many languages support consumes 1 byte of memory (regardless of the actual number of bits the data type requires).

Because the byte is the smallest unit of storage on most machines, and many languages use bytes to represent objects that require fewer than 8 bits, we need some way of denoting individual bits within a byte. To describe the bits within a byte, we'll use *bit numbers*. As Figure 2-3 shows, bit 0 is the *low-order (LO)*, or *least significant*, bit, and bit 7 is the *high-order (HO)*, or *most significant*, bit of the byte. We'll refer to all other bits by their number.

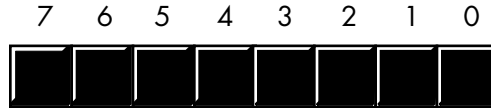


Figure 2-3: Bit numbering in a byte

A *word* is defined differently depending on the CPU: it may be a 16-bit, 32-bit, or 64-bit object. This book adopts the 80x86 terminology and defines a word as a collection of 16 bits. As with bytes, we'll use bit numbers for a word, starting with bit number 0 for the LO bit and working our way up to bit 15, the HO bit (see Figure 2-4).



Figure 2-4: Bit numbers in a word

Notice that a word contains exactly 2 bytes. Bits 0 through 7 form the LO byte, and bits 8 through 15 form the HO byte (see Figure 2-5).

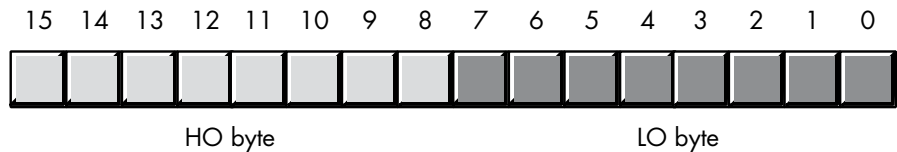


Figure 2-5: The 2 bytes in a word

A *double word* (or *dword*) is exactly what its name implies—a pair of words. Therefore, a double-word quantity is 32 bits long, as shown in Figure 2-6.

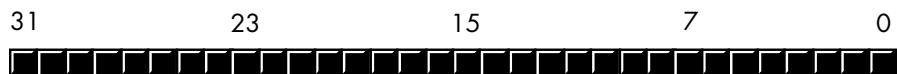


Figure 2-6: Bit layout in a double word

Figure 2-7 shows that a double word comprises 2 words or 4 bytes.

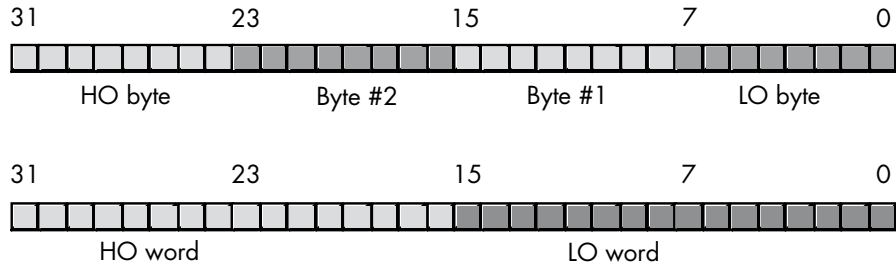


Figure 2-7: Bytes and words in a double word

As noted, most CPUs efficiently handle objects up to a certain size (typically 32 or 64 bits on contemporary systems). That doesn't mean you can't work with larger objects, only that it's less efficient to do so. You typically won't see programs handling numeric objects much larger than about 128 or 256 bits. Some programming languages make 64-bit integers available, and most languages support 64-bit floating-point values, so for these data types we'll use the term *quad word*. Finally, we'll use *long word* to describe 128-bit values; although few languages today support them,³ this gives us some room to grow.

We can break down quad words into 2 double words, 4 words, 8 bytes, or 16 nibbles. Likewise, we can break down long words into 2 quad words, 4 double words, 8 words, or 16 bytes.

Intel 80x86 platforms also support an 80-bit type that Intel calls a *tbyte* (short for "ten byte") object. The 80x86 CPU family uses tbyte variables to hold extended precision floating-point values and certain binary-coded decimal (BCD) values.

In general, with an n -bit string you can represent up to 2^n different values. Table 2-3 shows the number of possible objects you can represent with nibbles, bytes, words, double words, quad words, and long words.

Table 2-3: Number of Values Representable with Bit Strings

Size of bit string (in bits)	Number of possible combinations (2^n)
4	16
8	256
16	65,536
32	4,294,967,296
64	18,446,744,073,709,551,616
128	340,282,366,920,938,463,463,374,607,431,768,211,456

3. HLA supports 128-bit values.

2.5 Signed and Unsigned Numbers

The binary number $0\dots00000$ ⁴ represents 0; $0\dots00001$ represents 1; $0\dots00010$ represents 2; and so on toward infinity. But what about negative numbers? To represent signed values, most computer systems use the *two's complement* numbering system. The representation of signed numbers places some fundamental restrictions on them, so it's important that you understand how signed and unsigned numbers are represented differently in a computer system in order to use them efficiently.

With n bits, we can represent only 2^n different objects. Because negative values are objects in their own right, we'll have to divide these 2^n combinations between negative and non-negative values. So, for example, a byte can represent the negative values -128 through -1 and the non-negative values 0 to 127 . With a 16-bit word, we can represent signed values in the range $-32,768$ to $+32,767$. With a 32-bit double word, we can represent values in the range $-2,147,483,648$ to $+2,147,483,647$. In general, with n bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1} - 1$.

The two's complement system uses the HO bit as a *sign bit*. If the HO bit is 0, the number is non-negative and has the usual binary encoding; if the HO bit is 1, the number is negative and uses the two's complement encoding. Here are some examples using 16-bit numbers:

- $\$8000$ ($\%1000_0000_0000_0000$) is negative because the HO bit is 1.
- $\$100$ ($\%0000_0001_0000_0000$) is non-negative because the HO bit is 0.
- $\$7FFF$ ($\%0111_1111_1111_1111$) is non-negative.
- $\$FFFF$ ($\%1111_1111_1111_1111$) is negative.
- $\$FFF$ ($\%0000_1111_1111_1111$) is non-negative.

To negate a number, you can use the two's complement operation as follows:

1. Invert all the bits in the number; that is, change all the 0s to 1s and vice versa.
2. Add 1 to the inverted result (ignoring any overflow).

If the result is negative (has its HO bit set), then this is the two's complement form of the non-negative value.

For example, these are the steps to compute the 8-bit equivalent of the decimal value -5 :

1. $\%0000_0101$ 5 (in binary).
2. $\%1111_1010$ Invert all the bits.
3. $\%1111_1011$ Add 1 to obtain -5 (in two's complement form).

4. The ellipses (. . .) have the standard mathematical meaning: repeat a string of zeros an indefinite number of times.

If we take -5 and negate it, the result is 5 ($\%0000_0101$), just as we expect:

1. $\%1111_1011$ Two's complement for -5 .
2. $\%0000_0100$ Invert all the bits.
3. $\%0000_0101$ Add 1 to obtain 5 (in binary).

Let's look at some 16-bit examples and their negations.

First, negate $32,767$ ($\$7FFF$):

1. $\%0111_1111_1111_1111$ $+32,767$, the largest 16-bit positive number.
2. $\%1000_0000_0000_0000$ Invert all the bits ($8000h$).
3. $\%1000_0000_0000_0001$ Add 1 ($8001h$, or $-32,767$).

Now negate $16,384$ ($\$4000$):

1. $\%0100_0000_0000_0000$ $16,384$.
2. $\%1011_1111_1111_1111$ Invert all the bits ($\$BFFF$).
3. $\%1100_0000_0000_0000$ Add 1 ($\$C000$ or $-16,384$).

And now negate $-32,768$ ($\$8000$):

1. $\%1000_0000_0000_0000$ $-32,768$, the smallest 16-bit negative number.
2. $\%0111_1111_1111_1111$ Invert all the bits ($\$7FFF$).
3. $\%1000_0000_0000_0000$ Add 1 ($\$8000$ or $-32,768$).

$\$8000$ inverted becomes $\$7FFF$, and after adding 1 we obtain $\$8000$!

Wait, what's going on here: $-(-32,768)$ is $-32,768$? Of course not. However, the 16-bit two's complement numbering system cannot represent the value $+32,768$. In general, you cannot negate the smallest negative value in the two's complement numbering system.

2.6 Useful Properties of Binary Numbers

Here are some properties of binary values that you might find useful in your programs:

- If bit position 0 of a binary (integer) value contains 1 , the number is an odd number; if this bit contains 0 , then the number is even.
- If the LO n bits of a binary number all contain 0 , then the number is evenly divisible by 2^n .
- If a binary value contains a 1 in bit position n , and 0 s everywhere else, then that number is equal to 2^n .
- If a binary value contains all 1 s from bit position 0 up to (but not including) bit position n , and all other bits are 0 , then that value is equal to $2^n - 1$.

- Shifting all the bits in a number to the left by one position multiplies the binary value by 2.
- Shifting all the bits of an unsigned binary number to the right by one position effectively divides that number by 2 (this does not apply to signed integer values). Odd numbers are rounded down.
- Multiplying two n -bit binary values together may require as many as $2 \times n$ bits to hold the result.
- Adding or subtracting two n -bit binary values never requires more than $n + 1$ bits to hold the result.
- Inverting all the bits in a binary number (that is, changing all the 0s to 1s and vice versa) is the same thing as negating (changing the sign) of the value and then subtracting 1 from the result.
- *Incrementing* (adding 1 to) the largest unsigned binary value for a given number of bits always produces a value of 0.
- *Decrementing* (subtracting 1 from) 0 always produces the largest unsigned binary value for a given number of bits.
- An n -bit value provides 2^n unique combinations of those bits.
- The value $2^n - 1$ contains n bits, each containing the value 1.

It's a good idea to memorize all the powers of 2 from 2^0 through 2^{16} (see Table 2-4), as these values come up in programs all the time.

Table 2-4: Powers of 2

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536

2.7 Sign Extension, Zero Extension, and Contraction

With the two's complement system, a single negative value is represented differently depending on the size of the representation. An 8-bit signed value must be converted for use in an expression involving a 16-bit number. This conversion and its converse—converting a 16-bit value to 8 bits—are the *sign extension* and *contraction* operations, respectively.

Consider the value -64 . The 8-bit two's complement value for this number is $\$C0$. The 16-bit equivalent is $\$FFC0$. Clearly, these are not the same bit pattern. Now consider the value $+64$. The 8- and 16-bit versions of this value are $\$40$ and $\$0040$, respectively. We extend the size of negative values differently than we extend the size of non-negative values.

To *sign-extend* a value, copy the sign bit into the additional HO bits in the new format. For example, to sign-extend an 8-bit number to a 16-bit number, copy bit 7 of the 8-bit number into bits 8 through 15 of the 16-bit number. To sign-extend a 16-bit number to a double word, copy bit 15 into bits 16 through 31 of the double word.

When adding a byte quantity to a word quantity, you need to sign-extend the byte to 16 bits before adding the two numbers. Other operations may require a sign extension to 32 bits.

Table 2-5 provides several examples of sign extension.

Table 2-5: Sign Extension Examples

8 bits	16 bits	32 bits	Binary (two's complement)
$\$80$	$\$FF80$	$\$FFFF_FF80$	%1111_1111_1111_1111_1111_1000_0000
$\$28$	$\$0028$	$\$0000_0028$	%0000_0000_0000_0000_0000_0000_0010_1000
$\$9A$	$\$FF9A$	$\$FFFF_FF9A$	%1111_1111_1111_1111_1111_1001_1010
$\$7F$	$\$007F$	$\$0000_007F$	%0000_0000_0000_0000_0000_0000_0111_1111
n/a	$\$1020$	$\$0000_1020$	%0000_0000_0000_0000_0001_0000_0010_0000
n/a	$\$8086$	$\$FFFF_8086$	%1111_1111_1111_1111_1000_0000_1000_0110

Zero extension converts small unsigned values to larger unsigned values. Zero extension is very easy—just store 0s in the HO byte(s) of the larger operand. For example, to zero-extend the 8-bit value $\$82$ to 16 bits, you insert a 0 for the HO byte, yielding $\$0082$.

Further examples are listed in Table 2-6.

Table 2-6: Zero Extension Examples

8 bits	16 bits	32 bits	Binary
$\$80$	$\$0080$	$\$0000_0080$	%0000_0000_0000_0000_0000_0000_1000_0000
$\$28$	$\$0028$	$\$0000_0028$	%0000_0000_0000_0000_0000_0000_0010_1000
$\$9A$	$\$009A$	$\$0000_009A$	%0000_0000_0000_0000_0000_0000_1001_1010
$\$7F$	$\$007F$	$\$0000_007F$	%0000_0000_0000_0000_0000_0000_0111_1111
n/a	$\$1020$	$\$0000_1020$	%0000_0000_0000_0000_0001_0000_0010_0000
n/a	$\$8086$	$\$0000_8086$	%0000_0000_0000_0000_1000_0000_1000_0110

Many high-level language compilers automatically handle sign and zero extension. The following examples in C demonstrate how this works:

```
signed char sbyte;    // Chars in C are byte values.
short int sword;      // Short integers in C are *usually* 16-bit values.
long int sdword;      // Long integers in C are *usually* 32-bit values.
. . .
sword = sbyte;         // Automatically sign-extends the 8-bit value to 16 bits.
sdword = sbyte;        // Automatically sign-extends the 8-bit value to 32 bits.
sdword = sword;        // Automatically sign-extends the 16-bit value to 32 bits.
```

Some languages (such as Ada or Swift) require an explicit cast from a smaller size to a larger size. Check the reference manual for your particular language to see if this is necessary. The advantage of a language that requires you to provide an explicit conversion is that the compiler never does anything behind your back. If you fail to do the conversion yourself, the compiler emits a diagnostic message.

The important thing to realize about sign and zero extension is that they aren't always free. Assigning a smaller integer to a larger integer may require more machine instructions (taking longer to execute) than moving data between two like-sized integer variables. Therefore, you should be careful about mixing variables of different sizes within the same arithmetic expression or assignment statement.

Sign contraction—converting a value with some number of bits to the same value with a fewer number of bits—is a little more troublesome. For example, consider the value -448 . As a 16-bit hexadecimal number, its representation is $\$FE40$. The magnitude of this number is too large to fit into 8 bits, so you can't sign-contract it to 8 bits.

To properly sign-contract one value to another, you must look at the HO byte(s) that you want to discard. First, the HO bytes must all contain either 0 or $\$FF$. Second, the HO bit of your resulting value must match *every* bit you've removed from the number. Here are some examples of converting 16-bit values to 8-bit values (including a couple of failures):

- $\$FF80$ ($\%1111_1111_1000_0000$) can be sign-contracted to $\$80$ ($\%1000_0000$).
- $\$0040$ ($\%0000_0000_0100_0000$) can be sign-contracted to $\$40$ ($\%0100_0000$).
- $\$FE40$ ($\%1111_1110_0100_0000$) cannot be sign-contracted to 8 bits.
- $\$0100$ ($\%0000_0001_0000_0000$) cannot be sign-contracted to 8 bits.

Some high-level languages, like C, will simply store the LO portion of the expression into a smaller variable and discard the HO component—at best, the C compiler may give you a warning about the loss of precision that may occur. You can often quiet the compiler, but it still doesn't check for invalid values. Typically, you'd use code like the following to sign-contract a value in C:

```
signed char sbyte;    // Chars in C are byte values.
short int sword;      // Short integers in C are *usually* 16-bit values.
long int sdword;      // Long integers in C are *usually* 32-bit values.
```

```
. . .
sbyte = (signed char) sword;
sbyte = (signed char) sdword;
sword = (short int) sdword;
```

The only safe solution in C is to compare the result of the expression to an upper- and lower-bound value before attempting to store the value into a smaller variable. Here's the preceding code with these checks in place:

```
if( sword >= -128 && sword <= 127 )
{
    sbyte = (signed char) sword;
}
else
{
    // Report appropriate error.
}

// Another way, using assertions:

assert( sword >= -128 && sword <= 127 )
sbyte = (signed char) sword;

assert( sdword >= -32768 && sdword <= 32767 )
sword = (short int) sdword;
```

This code gets pretty ugly. In C/C++, you'd probably want to turn this into a macro (`#define`) or a function so your code would be a bit more readable.

Some high-level languages (such as Free Pascal and Delphi) automatically sign-convert values and then check the value to ensure it fits in the destination operand.⁵ Such languages raise some sort of exception (or stop the program) if a range violation occurs. To take corrective action, you'll either need to write some exception-handling code or use an `if` statement sequence similar to the one in the C example just given.

2.8 Saturation

You can also reduce the size of an integer value through *saturation*, which is useful when you're willing to live with a possible loss of precision. To convert a value via saturation, you copy the LO bits of the larger object into the smaller object. If the larger value is outside the smaller object's range, then you *clip* the larger value by setting the smaller object to the largest (or smallest) value within the smaller value's range.

For example, when converting a 16-bit signed integer to an 8-bit signed integer, if the 16-bit value is in the range -128 through +127, you simply copy the LO byte into the 8-bit object. If the 16-bit signed value is

5. Borland's compilers require the use of a special compiler directive to activate this check. By default, the compiler does not do the bounds check.

greater than +127, then you clip the value to +127 and store +127 into the 8-bit object. Likewise, if the value is less than -128, you clip the final 8-bit object to -128. Saturation works the same way when you clip 32-bit values to smaller values.

If the larger value is outside the range of the smaller value, there will be a loss of precision during the conversion. While clipping the value is never desirable, sometimes it's better than raising an exception or otherwise rejecting the calculation. For many applications, such as audio or video, the clipped result is still recognizable to the end user, so this is a reasonable conversion scheme.

Many CPUs support saturation arithmetic in their special “multimedia extension” instruction sets—for example, the MMX/SSE/AVX instruction extensions on the Intel 80x86 processor family. Most CPUs’ standard instruction sets, as well as most high-level languages, do not provide direct support for saturation, but the technique is not difficult. Consider the following Free Pascal/Delphi code, which uses saturation to convert a 32-bit integer to a 16-bit integer:

```
var
  li :longint;
  si :smallint;
  . . .
  if( li > 32767 ) then

    si := 32767;

  else if( li < -32768 ) then

    si := -32768;

  else
    si := li;
```

2.9 Binary-Coded Decimal Representation

The *binary-coded decimal (BCD)* format, as its name suggests, encodes decimal values using a binary representation. Common general-purpose high-level languages (like C/C++, Pascal, and Java) rarely support decimal values. However, business-oriented programming languages (like COBOL and many database languages) do. So, if you’re writing code that interfaces with a database or some language that supports decimal arithmetic, you may need to deal with BCD representation.

BCD values consist of a sequence of nibbles, with each nibble representing a value in the range 0 to 9. (The BCD format uses only 10 of the possible 16 values represented by a nibble.) With a byte we can represent values containing two decimal digits (0..99), as shown in Figure 2-8. With a word, we can represent four decimal digits (0..9999). A double word can represent up to eight decimal digits.

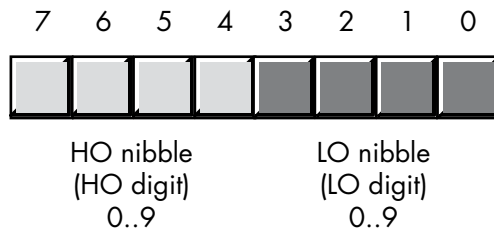


Figure 2-8: BCD data representation in a byte

An 8-bit BCD variable can represent values in the range 0 to 99, while that same 8 bits, holding a binary value, could represent values in the range 0 to 255. Likewise, a 16-bit binary value can represent values in the range 0 to 65,535, while a 16-bit BCD value can represent only about a sixth of those values (0..9999). Inefficient storage isn't the only problem with BCD, though—BCD calculations also tend to be slower than binary calculations.

The BCD format does have two saving graces: it's very easy to convert BCD values between the internal numeric representation and their decimal string representations, and it's also very easy to encode multidigit decimal values in hardware when using BCD—for example, when using a set of dials, with each dial representing a single digit. For these reasons, you're likely to see people using BCD in embedded systems (such as toaster ovens and alarm clocks) but rarely in general-purpose computer software.

A few decades ago, people thought that calculations involving BCD (or just decimal) arithmetic were more accurate than binary calculations. Therefore, they would often perform important calculations, like those involving monetary units, using decimal-based arithmetic. Certain calculations can produce more accurate results in BCD, but for most calculations, binary is more accurate. This is why most modern computer programs represent all values (including decimal values) in a binary form. For example, the Intel 80x86 *floating-point unit (FPU)* supports a pair of instructions for loading and storing BCD values. Internally, the FPU converts these BCD values to binary. It only uses BCD as an external (to the FPU) data format. This approach generally produces more accurate results.

2.10 Fixed-Point Representation

There are two ways computer systems commonly represent numbers with fractional components: fixed-point representation and floating-point representation.

Back in the days when CPUs didn't support floating-point arithmetic in hardware, fixed-point arithmetic was very popular with programmers writing high-performance software that dealt with fractional values. There's less software overhead needed to support fractional values in a fixed-point format than in floating-point. However, CPU manufacturers added FPUs to their CPUs to support floating-point in hardware, and today, it's fairly rare

to see someone attempt fixed-point arithmetic on a general-purpose CPU. It's usually more cost-effective to use the CPU's native floating-point format.

Although CPU manufacturers have worked hard at optimizing the floating-point arithmetic on their systems, in certain circumstances, carefully written assembly language programs that use fixed-point calculations will run faster than the equivalent floating-point code. Certain 3D gaming applications, for example, may produce faster computations using a 16:16 (16-bit integer, 16-bit fractional) format rather than a 32-bit floating-point format. Because there are some very good uses for fixed-point arithmetic, this section discusses fixed-point representation and fractional values using the fixed-point format.

NOTE

Chapter 4 will discuss the floating-point format.

As you've seen, positional numbering systems represent fractional values (values between 0 and 1) by placing digits to the right of the radix point. In the binary numbering system, each bit to the right of the binary point represents the value 0 or 1 multiplied by some successive negative power of 2. We represent that fractional component of a value using sums of binary fractions. For example, the value 5.25 is represented by the binary value 101.01. The conversion to decimal yields:

$$1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-2} = 4 + 1 + 0.25 = 5.25$$

When using a fixed-point binary format, you choose a particular bit in the binary representation and implicitly place the binary point before that bit. You choose the position of the binary point based on the number of significant bits you require in the fractional portion of the number. For example, if your values' integer components can range from 0 to 999, you'll need at least 10 bits to the left of the binary point to represent this range of values. If you require signed values, you'll need an extra bit for the sign. In a 32-bit fixed-point format, this leaves either 21 or 22 bits for the fractional part, depending on whether your value is signed.

Fixed-point numbers are a small subset of the real numbers. Because the number of values between any two integer values is infinite, fixed-point values cannot exactly represent every single one (doing so would require an infinite number of bits). With fixed-point representation, we have to approximate most of the real numbers. Consider the 8-bit fixed-point format, which uses 6 bits for the integer portion and 2 bits for the fractional component. The integer component can represent values in the range 0 to 63 (or signed values in the range -32 to +31). The fractional component can represent only four different values: 0.0, 0.25, 0.5, and 0.75. You cannot exactly represent 1.3 with this format; the best you can do is approximate it by choosing the value closest to it (1.25). This introduces error. You can reduce this error by adding further bits to the right of the binary point in your fixed-point format (at the expense of reducing the range of the integer component or adding more bits to your fixed-point format). For example, if you move to a 16-bit fixed-point format using an 8-bit integer

and an 8-bit fractional component, then you can approximate 1.3 using the binary value 1.01001101. The decimal equivalent is as follows:

$$1 + 0.25 + 0.03125 + 0.15625 + 0.00390625 = 1.30078125$$

Adding more bits to the fractional component of your fixed-point number will give you a more accurate approximation of this value (the error is only 0.00078125 using this format, compared to 0.05 in the previous format).

In a fixed-point binary numbering system, there are certain values you can never accurately represent regardless of how many bits you add to the fractional part of your fixed-point representation (1.3 just happens to be such a value). This is probably the main reason why people (mistakenly) feel that decimal arithmetic is more accurate than binary arithmetic (particularly when working with decimal fractions like 0.1, 0.2, 0.3, and so on).

To contrast the comparative accuracy of the two systems, let's consider a fixed-point decimal system (using BCD representation). If we choose a 16-bit format with 8 bits for the integer portion and 8 bits for the fractional portion, we can represent decimal values in the range 0.0 to 99.99 with two decimal digits of precision to the right of the decimal point. We can exactly represent values like 1.3 in this BCD notation using a hex value like \$0130 (the implicit decimal point appears between the second and third digits in this number). As long as you use only the fractional values 0.00 to 0.99 in your computations, this BCD representation is more accurate than the binary fixed-point representation (using an 8-bit fractional component).

In general, however, the binary format is more accurate. The binary format lets you exactly represent 256 different fractional values, whereas BCD lets you represent only 100. If you pick an arbitrary fractional value, it's likely the binary fixed-point representation provides a better approximation than the decimal format (because there are over two and a half times as many binary versus decimal fractional values). (You can extend this comparison to larger formats: for example, with a 16-bit fractional component, the decimal/BCD fixed-point format gives you exactly four digits of precision; the binary format, on the other hand, offers over six times the resolution—65,536 rather than 10,000 fractional values.) Decimal fixed-point format has the advantage only when you regularly work with the fractional values that it can exactly represent. In the United States, monetary computations commonly produce these fractional values, so programmers figured the decimal format is better for monetary computations. However, given the accuracy most financial computations require (generally four digits to the right of the decimal point is the minimum precision), it's usually better to use a binary format.

If you absolutely, positively need to exactly represent the fractional values between 0.00 and 0.99 with at least two digits of precision, the binary fixed-point format is not a viable solution. Fortunately, you don't have to use a decimal format; as you'll soon see, there are other binary formats that will let you exactly represent these values.

2.11 Scaled Numeric Formats

Fortunately, there's a numeric representation that combines the exact representation of certain decimal fractions with the precision of the binary format. Known as the *scaled numeric* format, this representation is also efficient to use and doesn't require any special hardware.

Another advantage of the scaled numeric format is that you can choose any base, not just decimal, for your format. For example, if you're working with ternary (base-3) fractions, you can multiply your original input value by 3 (or a power of 3) and exactly represent values like $\frac{1}{3}$, $\frac{2}{3}$, $\frac{4}{9}$, $\frac{7}{27}$, and so on—something you can't do in either the binary or decimal numbering systems.

To represent fractional values, you multiply your original value by some value that converts the fractional component to a whole number. For example, if you want to maintain two decimal digits of precision to the right of the decimal point, multiply your values by 100 upon input. This translates values like 1.3 to 130, which we can exactly represent using an integer value. Assuming you do this calculation with all your fractional values (and they have the same two digits of precision to the right of the decimal point), you can manipulate your values using standard integer arithmetic operations. For example, if you have the values 1.5 and 1.3, their integer conversion produces 150 and 130. If you add these two values, you get 280 (which corresponds to 2.8). When you need to output these values, you divide them by 100 and emit the quotient as the integer portion of the value and the remainder (zero-extended to two digits, if necessary) as the fractional component. Other than needing to write specialized input and output routines that handle the multiplication and division by 100 (as well as dealing with the decimal point), you'll find that this scaled numeric scheme is almost as easy as doing regular integer calculations.

If you scale your values as described here, you've limited the maximum range of the integer portion of your numbers. For example, if you need two decimal digits of precision to the right of your decimal point (meaning you multiply the original value by 100), then you may only represent (unsigned) values in the range 0 to 42,949,672 rather than the normal range of 0 to 4,294,967,296.

When you're doing addition or subtraction with a scaled format, both operands must have the same scaling factor. If you've multiplied the left operand by 100, you must multiply the right operand by 100 as well. For example, if you've scaled the variable `i10` by 10 and you've scaled the variable `j100` by 100, you need to either multiply `i10` by 10 (to scale it by 100) or divide `j100` by 10 (to scale it down to 10) before attempting to add or subtract these two numbers. This ensures that both operands have the radix point in the same position (note that this applies to literal constants as well as to variables).

In multiplication and division operations, the operands do not require the same scaling factor prior to the operation. However, once the operation is complete, you may need to adjust the result. Suppose you have two values you've scaled by 100 to produce two digits of precision after the decimal point, $i = 25$ (0.25) and $j = 1$ (0.01). If you compute $k = i * j$ using standard integer arithmetic, you'll get 25 ($25 \times 1 = 25$), which is interpreted as 0.25, but the result should be 0.0025. The computation is correct; the problem is understanding how the multiplication operator works. We're actually computing:

$$\begin{aligned} & (0.25 \times (100)) \times (0.01 \times (100)) \\ & = \\ & 0.25 \times 0.01 \times (100 \times 100) \text{ (commutative laws allow this)} \\ & = \\ & 0.0025 \times (10,000) \\ & = \\ & 25 \end{aligned}$$

The final result actually gets scaled by 10,000 because both i and j have been multiplied by 100; when you multiply their values, you wind up with a value multiplied by 10,000 (100×100) rather than 100. To solve this problem, you should divide the result by the scaling factor once the computation is complete. For example, $k = (i * j)/100$.

The division operation suffers from a similar problem. Suppose we have the values $m = 500$ (5.0) and $n = 250$ (2.5) and we want to compute $k = m/n$. We would normally expect to get the result 200 (2.0, which is $5.0/2.5$). However, here's what we're actually computing:

$$\begin{aligned} & (5 \times 100) / (2.5 \times 100) \\ & = \\ & 500/250 \\ & = \\ & 2 \end{aligned}$$

At first blush this may look correct, but the result is really 0.02 after you factor in the scaling operation. The result we need is 200 (2.0). Division by the scaling factor eliminates the scaling factor in the final result. Therefore, to properly compute the result, we need to compute $k = 100 * m/n$.

Multiplication and division place a limit on the precision you have available. If you have to premultiply the dividend by 100, then the dividend must be at least 100 times smaller than the largest possible integer value, or an overflow will occur (producing an incorrect result). Likewise, when you're multiplying two scaled values, the final result must be 100 times less than the maximum integer value, or an overflow will occur. Because of these issues, you may need to set aside additional bits or work with small numbers when using scaled numeric representation.

2.12 Rational Representation

One big problem with the fractional representations we've seen is that they provide a close approximation, but not an exact representation, for all rational values.⁶ For example, in binary or decimal you cannot exactly represent the value $\frac{1}{3}$. You could switch to a ternary (base-3) numbering system and exactly represent $\frac{1}{3}$, but then you wouldn't be able to exactly represent fractional values like $\frac{1}{2}$ or $\frac{1}{10}$. We need a numbering system that can represent *any* rational fractional value.

Rational representation uses pairs of integers to represent fractional values. One integer represents the numerator (n) of a fraction, and the other represents the denominator (d). The actual value is equal to n/d . As long as n and d are “relatively prime” (that is, not both evenly divisible by the same value), this scheme provides a good representation for fractional values within the bounds of the integer representation you're using for n and d . Arithmetic is quite easy; you use the same algorithms to add, subtract, multiply, and divide fractional values that you learned in grade school when dealing with fractions. However, certain operations may produce really large numerators or denominators (to the point where you get integer overflow in these values). Other than this problem, you can represent a wide range of fractional values using this scheme.

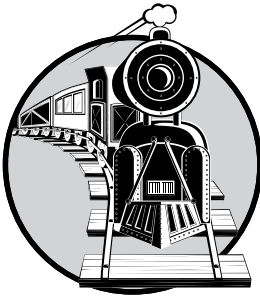
2.13 For More Information

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Boston: Addison-Wesley, 1998.

6. It isn't possible to provide an exact computer representation of an irrational number, so we won't even try.

3

BINARY ARITHMETIC AND BIT OPERATIONS



As Chapter 2 explained, understanding how computers represent data in binary is a prerequisite to writing software that works well on them. Of equal importance is understanding how computers operate on binary data. That's the focus of this chapter, which explores arithmetic, logical, and bit operations on binary data.

3.1 Arithmetic Operations on Binary and Hexadecimal Numbers

Often, you need to manually operate on two binary (or hexadecimal) values in order to use the result in your source code. Although there are calculators that can compute such results, you should be able to perform simple arithmetic operations on binary operands by hand. Hexadecimal arithmetic is sufficiently painful that a hexadecimal calculator (or a software-based calculator

that supports hexadecimal operations, such as the Windows calculator, or a smartphone app) belongs on every programmer's desk. Arithmetic operations on binary values, however, are easier than decimal arithmetic.

Knowing how to manually compute binary arithmetic results is essential because several important algorithms use these operations (or variants of them). This section describes how to manually add, subtract, multiply, and divide binary values, and how to perform various logical operations on them.

3.1.1 Adding Binary Values

Adding two binary values is easy; there are only eight rules to learn:¹

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ with carry
- Carry + $0 + 0 = 1$
- Carry + $0 + 1 = 0$ with carry
- Carry + $1 + 0 = 0$ with carry
- Carry + $1 + 1 = 1$ with carry

Once you know these eight rules, you can add any two binary values together. Here's a step-by-step example of binary addition:

```
  0101
+ 0011
-----
```

Step 1: Add the L0 bits ($1 + 1 = 0 + \text{carry}$).

```
    c
  0101
+ 0011
-----
    0
```

Step 2: Add the carry plus the bits in bit position 1 ($\text{carry} + 0 + 1 = 0 + \text{carry}$).

```
    c
  0101
+ 0011
-----
   00
```

Step 3: Add the carry plus the bits in bit position 2 ($\text{carry} + 1 + 0 = 0 + \text{carry}$).

```
    c
  0101
+ 0011
-----
  000
```

1. This might sound like a lot, but you had to memorize approximately 200 rules for decimal addition!

Step 4: Add the carry plus the bits in bit position 3 (carry + 0 + 0 = 1).

```
  0101
+ 0011
-----
 1000
```

Here are some more examples:

1100_1101	1001_1111	0111_0111
+ 0011_1011	+ 0001_0001	+ 0000_1001
-----	-----	-----
1_0000_1000	1011_0000	1000_0000

3.1.2 Subtracting Binary Values

Like addition, binary subtraction has eight rules:

- $0 - 0 = 0$
- $0 - 1 = 1$ with a borrow
- $1 - 0 = 1$
- $1 - 1 = 0$
- $0 - 0 - \text{borrow} = 1$ with a borrow
- $0 - 1 - \text{borrow} = 0$ with a borrow
- $1 - 0 - \text{borrow} = 0$
- $1 - 1 - \text{borrow} = 1$ with a borrow

Here's a step-by-step example of binary subtraction:

```
  0101
- 0011
-----
Step 1: Subtract the L0 bits (1 - 1 = 0).
  0101
- 0011
-----
    0
Step 2: Subtract the bits in bit position 1 (0 - 1 = 1 + borrow).
  0101
- 0011
  b
-----
   10
Step 3: Subtract the borrow and the bits in bit position 2 (1 - 0 - b = 0).
  0101
- 0011
-----
  010
```

Step 4: Subtract the bits in bit position 3 ($0 - 0 = 0$).

```
  0101
- 0011
-----
  0010
```

Here are some more examples:

1100_1101	1001_1111	0111_0111
- 0011_1011	- 0001_0001	- 0000_1001
-----	-----	-----
1001_0010	1000_1110	0110_1110

3.1.3 *Multiplying Binary Values*

Multiplication of binary numbers is simple; it follows the same rules as decimal multiplication involving only 0s and 1s:

- $0 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$

Here's a step-by-step example of binary multiplication:

```
  1010
× 0101
-----
```

Step 1: Multiply the 10 bit of the multiplier times the multiplicand.

```
  1010
× 0101
-----
  1010    (1 × 1010)
```

Step 2: Multiply bit 1 of the multiplier times the multiplicand.

```
  1010
× 0101
-----
  1010    (1 × 1010)
  0000    (0 × 1010)
-----
  01010   (partial sum)
```

Step 3: Multiply bit 2 of the multiplier times the multiplicand.

```
  1010
× 0101
-----
 001010   (previous partial sum)
  1010    (1 × 1010)
-----
 110010   (partial sum)
```

Step 4: Multiply bit 3 of the multiplier times the multiplicand.

$$\begin{array}{r}
 1010 \\
 \times 0101 \\
 \hline
 110010 \quad (\text{previous partial sum}) \\
 0000 \quad (0 \times 1010) \\
 \hline
 0110010 \quad (\text{product})
 \end{array}$$

3.1.4 Dividing Binary Values

Binary division uses the same (longhand) division algorithm as decimal division. Figure 3-1 shows the steps in a decimal division problem.

$ \begin{array}{r} 2 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \end{array} $	(1) 12 goes into 34 two times.
$ \begin{array}{r} 28 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 9 \end{array} $	(2) Subtract 24 from 34 and drop down the 105.
$ \begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \\ \underline{96} \\ 0 \end{array} $	(3) 12 goes into 105 eight times.
$ \begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \\ \underline{96} \\ 0 \end{array} $	(4) Subtract 96 from 105 and drop down the 96.
$ \begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \\ \underline{96} \\ 0 \end{array} $	(5) 12 goes into 96 exactly eight times.
$ \begin{array}{r} 288 \\ 12 \overline{) 3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \\ \underline{96} \\ 0 \end{array} $	(6) Therefore, 12 goes into 3456 exactly 288 times.

Figure 3-1: Decimal division (3456/12)

This algorithm is easier in binary because at each step you don't have to guess how many times 12 goes into the remainder or multiply 12 by your guess to obtain the amount to subtract. At each step in the binary algorithm, the divisor goes into the remainder exactly zero or one times. For example, consider the division of 27 (11011) by 3 (11) shown in Figure 3-2.

$\begin{array}{r} 1 \\ 11 \overline{) 11011} \\ \underline{11} \end{array}$	11 goes into 11 one time.	$\begin{array}{r} 1 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \end{array}$	Subtract out the 11 and bring down the 0.
$\begin{array}{r} 10 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ 00 \end{array}$	11 goes into 00 zero times.	$\begin{array}{r} 10 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ 00 \\ \underline{01} \end{array}$	Subtract out the 0 and bring down the 1.
$\begin{array}{r} 100 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ 00 \\ 01 \\ 00 \end{array}$	11 goes into 01 zero times.	$\begin{array}{r} 100 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ 00 \\ 01 \\ 00 \\ \underline{11} \end{array}$	Subtract out the 0 and bring down the 1.
$\begin{array}{r} 1001 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ 00 \\ 01 \\ 00 \\ \underline{11} \\ 11 \end{array}$	11 goes into 11 one time.	$\begin{array}{r} 1001 \\ 11 \overline{) 11011} \\ \underline{11} \\ 00 \\ 00 \\ 01 \\ 00 \\ \underline{11} \\ 11 \\ \underline{11} \\ 00 \end{array}$	This produces the final result of 1001.

Figure 3-2: Longhand division in binary

3.2 Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND, OR, XOR (exclusive-or), and NOT. In contrast to the arithmetic operations, a hexadecimal calculator isn't necessary to perform these operations.

The logical AND, OR, and XOR operations accept two single-bit operands and compute the following results:

AND:

0 and 0 = 0
 0 and 1 = 0
 1 and 0 = 0
 1 and 1 = 1

OR:

0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1

XOR:

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0

Tables 3-1, 3-2, and 3-3 show the *truth tables* for the AND, OR, and XOR operations, respectively. A truth table is just like the multiplication tables you encountered in elementary school. The values in the left column correspond to the left operand of the operation. The values in the top row correspond to the right operand. The result is at the intersection of the row and column (for a particular pair of operands).

Table 3-1: AND Truth Table

AND	0	1
0	0	0
1	0	1

Table 3-2: OR Truth Table

OR	0	1
0	0	1
1	1	1

Table 3-3: XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In plain English, the logical AND operation translates as, “If the first operand is 1 and the second operand is 1, the result is 1; otherwise the result is 0.” We could also state this as “If either or both operands are 0, the result is 0.” The logical AND operation is useful for forcing a 0 result. If one of the operands is 0, the result is 0 regardless of the value of the other operand. If one of the operands contains 1, then the result is the value of the other operand.

Colloquially, the logical OR operation is, “If the first operand or the second operand (or both) is 1, the result is 1; otherwise the result is 0.” This is also known as the *inclusive-OR* operation. If one of the operands to the logical-OR operation is 1, the result is 1. If an operand is 0, the result is the value of the other operand.

In English, the logical XOR operation is, “If the first or second operand, but not both, is 1, the result is 1; otherwise, the result is 0.” If one of the operands is a 1, the result is the *inverse* of the other operand.

The logical NOT operation is *unary* (meaning it accepts only one operand). Table 3-4 is the truth table for the NOT operation. This operator inverts the value of its operand.

Table 3-4: NOT Truth Table

NOT	0	1
	1	0

3.3 Logical Operations on Binary Numbers and Bit Strings

Because most programming languages manipulate groups of 8, 16, 32, or 64 bits, we need to extend the definition of these logical operations beyond single-bit operands to operate on a bit-by-bit (or *bitwise*) basis. Given two values, a bitwise logical function operates on bit 0 from both source operands, producing bit 0 in the result operand; it operates on bit 1 of both operands, producing bit 1 of the result; and so on. For example, if you want to compute the bitwise logical AND of two 8-bit numbers, you would logically AND each pair of bits in the two numbers:

```
%1011_0101
%1110_1110
-----
%1010_0100
```

This bit-by-bit execution applies to the other logical operations as well. The ability to force bits to 0 or 1 using the logical AND and OR operations, and to invert bits using the logical XOR operation, is very important when you’re working with strings of bits (such as binary numbers). These operations let you selectively manipulate certain bits within a value while leaving other bits unaffected. For example, if you have an 8-bit binary value *X* and you want to guarantee that bits 4 through 7 contain 0s, AND the value *X* with the binary value %0000_1111. This bitwise AND operation forces the HO 4 bits of *X* to 0 and leaves the LO 4 bits of *X* unchanged. Likewise, you could force the LO bit of *X* to 1 and invert bit number 2 of *X* by ORing *X* with %0000_0001 and then exclusive-ORing (XORing) *X* with %0000_0100.

Manipulating bit strings with the logical AND, OR, and XOR operations is known as *masking*. This term originates from the fact that we can use certain values (1 for AND, 0 for OR and XOR) to “mask out” or “mask in” certain bits in an operand while forcing other bits to 0, 1, or their inverse.

Several languages provide operators that let you compute the bitwise AND, OR, XOR, and NOT of their operands. The C/C++/Java/Swift language family uses the ampersand (&) for bitwise AND, the pipe (|) for

bitwise OR, the caret (^) for bitwise XOR, and the tilde (~) for bitwise NOT, as shown here:

```
// Here's a C/C++ example:

i = j & k;    // Bitwise AND
i = j | k;    // Bitwise OR
i = j ^ k;    // Bitwise XOR
i = ~j;       // Bitwise NOT
```

The Visual Basic and Free Pascal/Delphi languages let you use the `and`, `or`, `xor`, and `not` operators with integer operands. From 80x86 assembly language, you can use the `AND`, `OR`, `NOT`, and `XOR` instructions.

3.4 Useful Bit Operations

Although bit operations may seem a bit abstract, they are quite useful for many non-obvious purposes. This section describes some of their useful properties in various languages.

3.4.1 Testing Bits in a Bit String Using AND

You can use the bitwise AND operator to test individual bits in a bit string to see if they are 0 or 1. If you logically AND a value with a bit string that contains a 1 in a certain bit position, the result of the AND will be 0 if the corresponding bit contains a 0, and nonzero if that bit position contains 1. Consider the following C/C++ code, which checks an integer value to see if it is odd or even by testing bit 0 of the integer:

```
IsOdd = (ValueToTest & 1) != 0;
```

In binary form, here's what this bitwise AND operation is doing:

```
xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx // Assuming ValueToTest is 32 bits
0000_0000_0000_0000_0000_0000_0000_0001 // Bitwise AND with the value 1
-----
0000_0000_0000_0000_0000_0000_0000_000x // Result of bitwise AND
```

The result is 0 if the LO bit of *ValueToTest* contains a 0 in bit position 0. The result is 1 if *ValueToTest* contains a 1 in bit position 1. This calculation ignores all other bits in *ValueToTest*.

3.4.2 Testing a Set of Bits for Zero/Not Zero Using AND

You can also use the bitwise AND operator to see if all bits in a set are 0. For example, one way to check if a number is evenly divisible by 16 is to see if the LO 4 bits are all 0s. The following Free Pascal/Delphi statement uses the bitwise AND operator to accomplish this:

```
IsDivisibleBy16 := (ValueToTest and $f) = 0;
```

In binary form, here's what this bitwise AND operation is doing:

```
xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx // Assuming ValueToTest is 32 bits
0000_0000_0000_0000_0000_0000_0000_1111 // Bitwise AND with $F
-----
0000_0000_0000_0000_0000_0000_0000_xxxx // Result of bitwise AND
```

The result is 0 if and only if the LO 4 bits of ValueToTest are all 0.

3.4.3 Comparing a Set of Bits Within a Binary String

The AND and OR operations are particularly useful if you need to compare a subset of the bits in a binary value against some other value. For example, you might want to compare two 6-bit values found in bits 0, 1, 10, 16, 24, and 31 of a pair of 32-bit values. The trick is to set all the uninteresting bits to 0 and then compare the two results.²

Consider the following three binary values; x denotes bits whose values we don't care about:

```
%1xxxxxx0xxxxxxxx1xxxxx0xxxxxxxxx10
%1xxxxxx0xxxxxxxx1xxxxx0xxxxxxxxx10
%1xxxxxx1xxxxxxxx1xxxxx1xxxxxxxxx11
```

The first and second binary values (assuming we're interested only in bits 31, 24, 16, 10, 1, and 0) are equal. If we compare either of the first two values against the third value, we'll find that they are not equal. The third value is also greater than the first two. In C/C++ and assembly, this is how we could compare these values:

// C/C++ example

```
if( (value1 & 0x81010403) == (value2 & 0x81010403))
{
    // Do something if bits 31, 24, 16, 10, 1, and 0 of
    // value1 and value2 are equal
}

if( (value1 & 0x81010403) != (value3 & 0x81010403))
{
    // Do something if bits 31, 24, 16, 10, 1, and 0 of
    // value1 and value3 are not equal
}
```

// HLA/x86 assembly example:

```
mov( value1, eax );    // EAX = value1
and( $8101_0403, eax ); // Mask out unwanted bits in EAX
mov( value2, edx );    // EDX = value2
and( $8101_0403, edx ); // Mask out the same set of unwanted bits in EDX
```

2. It's also possible to set all the uninteresting bits to 1s via the OR operation, but the AND operator is often more convenient.

```

if( eax = edx ) then      // See if the remaining bits match

    // Do something if bits 31, 24, 16, 10, 1, and 0 of
    // value1 and value2 are equal

endif;

mov( value1, eax );      // EAX = value1
and( $8101_0403, eax );  // Mask out unwanted bits in EAX
mov( value3, edx );      // EDX = value2
and( $8101_0403, edx );  // Mask out the same set of unwanted bits in EDX

if( eax <> edx ) then     // See if the remaining bits do not match

    // Do something if bits 31, 24, 16, 10, 1, and 0 of
    // value1 and value3 are not equal

endif;

```

3.4.4 Creating Modulo- n Counters Using AND

A *modulo- n counter* counts from 0^3 to some maximum value and then resets to 0. Modulo- n counters are great for creating repeating sequences of numbers such as 0, 1, 2, 3, 4, 5, $\dots n-1$; 0, 1, 2, 3, 4, 5, $\dots n-1$; 0, 1, $\dots$. You can use such sequences to create circular queues and other objects that reuse array elements upon encountering the end of the data structure. The normal way to create a modulo- n counter is to add 1 to the counter, divide the result by n , and then keep the remainder. The following code examples demonstrate the implementation of a modulo- n counter in C/C++, Pascal, and Visual Basic:

```

cntr = (cntr + 1 ) % n;    // C/C++/Java/Swift
cntr := (cntr + 1) mod n;  // Pascal/Delphi
cntr = (cntr + 1) Mod n    ' Visual Basic

```

However, division is an expensive operation, requiring far more time to execute than addition. In general, you'll find it more efficient to implement modulo- n counters using a comparison rather than the remainder operator. Here's a Pascal example:

```

cntr := cntr + 1;          // Pascal example
if( cntr >= n ) then
    cntr := 0;

```

For certain special cases, when n is a power of 2, you can increment a modulo- n counter more efficiently and conveniently using the AND operation. To do so, increment your counter and then logically AND it with the value $x = 2^m - 1$ ($2^m - 1$ contains 1s in bit positions $0..m-1$, and 0s

3. Actually, they could count down to 0 as well, but usually they count up.

everywhere else). Because the AND operation is much faster than division, AND-driven modulo- n counters are much more efficient than those using the remainder operator. On most CPUs, using the AND operator is quite a bit faster than using an if statement. The following examples show how to implement a modulo- n counter for $n = 32$ using the AND operation:

//Note: $0x1f = 31 = 2^5 - 1$, so $n = 32$ and $m = 5$

```
cntr = (cntr + 1) & 0x1f;    // C/C++/Java/Swift example
cntr := (cntr + 1) and $1f;  // Pascal/Delphi example
cntr = (cntr + 1) and &h1f   ' Visual Basic example
```

The assembly language code is especially efficient:

```
inc( eax );                // Compute (eax + 1) mod 32
and( $1f, eax );
```

3.5 Shifts and Rotates

Another set of logical operations on bit strings are the *shift* and *rotate* operations. These functions can be further broken down into *shift lefts*, *rotate lefts*, *shift rights*, and *rotate rights*. These operations are very useful in many programs.

The shift left operation moves each bit in a bit string one position to the left, as shown in Figure 3-3. Bit 0 moves into bit position 1, the previous value in bit position 1 moves into bit position 2, and so on.

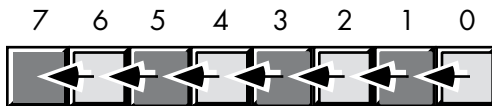


Figure 3-3: Shift left operation (on a byte)

You might be asking two questions: “What goes into bit 0?” and “Where does the HO bit wind up?” We’ll shift a 0 into bit 0, and the previous value of the HO bit will be the *carry* out of this operation.

Several high-level languages (such as C/C++/C#, Swift, Java, and Free Pascal/Delphi) provide a shift left operator. In the C language family, this operator is `<<`. In Free Pascal/Delphi, you use the `shl` operator. Here are some examples:

```
// C:
    cLang = d << 1;    // Assigns d shifted left one position to
                       // variable "cLang"

// Delphi:
    Delphi := d shl 1; // Assigns d shifted left one position to
                       // variable "Delphi"
```

Shifting the binary representation of a number one position to the left is equivalent to multiplying that value by 2. If you're using a programming language that doesn't provide an explicit shift left operator, you can simulate this by multiplying a binary integer value by 2. Although the multiplication operation is usually slower than the shift left operation, most compilers are smart enough to translate a multiplication by a constant power of 2 into a shift left operation. Therefore, you could write code like the following in Visual Basic to do a shift left:

```
vb = d * 2
```

A shift right operation is similar to a shift left, except we move the data in the opposite direction. Bit 7 moves into bit 6, bit 6 moves into bit 5, bit 5 moves into bit 4, and so on. During a shift right, we'll move a 0 into bit 7, and bit 0 will be the carry out of the operation (see Figure 3-4). C, C++, C#, Swift, and Java use the `>>` operator for a shift right operation. Free Pascal/Delphi uses the `shr` operator. Most assembly languages also provide a shift right instruction (`shr` on the 80x86).

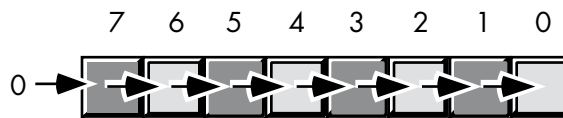


Figure 3-4: The shift right operation (on a byte)

Shifting an unsigned binary value one position to the right divides that value by 2. For example, if you shift the unsigned representation of 254 (`$FE`) one place to the right, you get 127 (`$7F`), exactly as you'd expect. However, if you shift the 8-bit two's complement binary representation of -2 (`$FE`) one position to the right, you get 127 (`$7F`), which is *not* correct. To divide a signed number by 2 using a shift, we use a third shift operation, *arithmetic shift right*, which doesn't modify the value of the HO bit. Figure 3-5 shows the arithmetic shift right operation for an 8-bit operand.

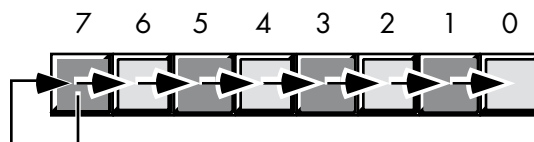


Figure 3-5: Arithmetic shift right operation (on a byte)

This generally produces the result you expect for two's complement signed operands. For example, if you perform the arithmetic shift right operation on -2 (`$FE`), you get -1 (`$FF`). Note, however, that this operation always rounds the numbers to the closest integer that is *less than or equal to the actual result*. If you arithmetically shift right -1 (`$FF`), the result is -1, not 0. Because -1 is less than 0, the arithmetic shift right operation rounds toward -1. This is not a “bug” in the arithmetic shift right operation; it just

uses a different (though valid) definition of integer division. The bottom line is that you probably won't be able to use a signed division operator as a substitute for arithmetic shift right in languages that don't support arithmetic shift right, because most integer division operators round toward 0.

It's rare for a high-level language to support both the logical shift right and the arithmetic shift right. Worse still, the specifications for certain languages leave it up to the compiler's implementer to decide whether to use an arithmetic shift right or a logical shift right operation. Therefore, it's only safe to use the shift right operator on values whose HO bit will cause both forms of the shift right operation to produce the same result. To guarantee that a shift right is a logical shift right or an arithmetic shift right operation, you'll either have to drop down into assembly language or handle the HO bit manually. The high-level code gets ugly really fast, so a quick inline assembly statement might be a better solution if your program doesn't need to be portable across different CPUs. The following code demonstrates how to simulate a 32-bit logical shift right and arithmetic shift right in languages that don't guarantee the type of shift they use:

```
// Written in C/C++, assuming 32-bit integers, logical shift right:
// Compute bit 30.
Bit30 = ((ShiftThisValue & 0x80000000) != 0) ? 0x40000000 : 0;
// Shifts bits 0..30.
ShiftThisValue = (ShiftThisValue & 0x7fffffff) >> 1;
// Merge in Bit #30.
ShiftThisValue = ShiftThisValue | Bit30;

// Arithmetic shift right operation

Bits3031 = ((ShiftThisValue & 0x80000000) != 0) ? 0xC0000000 : 0;
// Shifts bits 0..30.
ShiftThisValue = (ShiftThisValue & 0x7fffffff) >> 1;
// Merge bits 30/31.
ShiftThisValue = ShiftThisValue | Bits3031;
```

Many assembly languages also provide various rotate instructions that circulate bits through an operand by taking the bits shifted out of one end of the operand and shifting them into the other end. Few high-level languages provide this operation; fortunately, you won't need it very often. If you do, you can synthesize this operation using the shift operators available in your high-level language:

```
// Pascal/Delphi Rotate Left, 32-bit example:
// Puts bit 31 into bit 0, clears other bits.
CarryOut := (ValueToRotate shr 31);
ValueToRotate := (ValueToRotate shl 1) or CarryOut;
```

For more information on the type of shift and rotate operations that are possible, consult *The Art of Assembly Language* (No Starch Press).

3.6 Bit Fields and Packed Data

CPUs generally operate most efficiently on byte, word, double-word and quad-word data types,⁴ but occasionally you'll need to work with a data type whose size is something other than 8, 16, 32, or 64 bits. In such cases, you may be able to save some memory by *packing* different strings of bits together as compactly as possible, without wasting any bits to align a particular data field on a byte or other boundary.

Consider a date of the form 04/02/01. It takes three numeric values to represent this date: month, day, and year. Months use the values 1 through 12, which require at least 4 bits to represent. Days use the range 1 through 31, which take 5 bits to represent. The year value, assuming that we're working with values in the range 0 through 99, requires 7 bits. The total of $4 + 5 + 7$ is 16 bits, or 2 bytes. We can pack our date data into 2 bytes rather than the 3 that would be required if we used a separate byte for each of the values. This saves 1 byte of memory for each date stored, which could be a substantial saving if you need to store many dates. You might arrange the bits as shown in Figure 3-6.



Figure 3-6: Short packed date format (16 bits)

MMMM represents the 4 bits that hold the month value, DDDDD the 5 bits that hold the day, and YYYYYYY the 7 bits that hold the year. Each collection of bits representing a data item is a *bit field*. We could represent April 2, 2001, with \$4101:

0100	00010	0000001		= %0100_0001_0000_0001 or \$4101
04	02	01		

Although packed values are space efficient (that is, they use little memory), they are computationally inefficient (slow!). The reason? It takes extra instructions to unpack the data from the various bit fields. These extra instructions take time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything. The following sample HLA/x86 code demonstrates packing and unpacking this 16-bit date format.

```
program dateDemo;

#include( "stdlib.hhf" )

static
```

4. Some RISC CPUs only operate efficiently on double-word or quad-word values, so the concept of bit fields and packed data may apply to any object less than 32 or even 64 bits in size on such CPUs.

```

day:      uns8;
month:    uns8;
year:     uns8;
packedDate: word;

```

```
begin dateDemo;
```

```

stdout.put( "Enter the current month, day, and year: " );
stdin.get( month, day, year );

```

```
// Pack the data into the following bits:
```

```

//
// 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
//  m m m m d d d d d y y y y y y y

```

```

mov( 0, ax );
mov( ax, packedDate ); // Just in case there is an error.
if( month > 12 ) then
    stdout.put( "Month value is too large", nl );

elseif( month = 0 ) then
    stdout.put( "Month value must be in the range 1..12", nl );

elseif( day > 31 ) then
    stdout.put( "Day value is too large", nl );

elseif( day = 0 ) then
    stdout.put( "Day value must be in the range 1..31", nl );

elseif( year > 99 ) then
    stdout.put( "Year value must be in the range 0..99", nl );

else

    mov( month, al );
    shl( 5, ax );
    or( day, al );
    shl( 7, ax );
    or( year, al );
    mov( ax, packedDate );

```

```
endif;
```

```

// Okay, display the packed value:
stdout.put( "Packed data = $", packedDate, nl );

```

```
// Unpack the date:
```

```

mov( packedDate, ax );
and( $7f, al ); // Retrieve the year value.
mov( al, year );

```

```

mov( packedDate, ax ); // Retrieve the day value.
shr( 7, ax );
and( %1_1111, al );
mov( al, day );

```



```

mov( packedDate, ax ); // Retrieve the month value.
rol( 4, ax );
and( %1111, al );
mov( al, month );

stdout.put( "The date is ", month, "/", day, "/", year, nl );

end dateDemo;

```

Keeping in mind the Y2K⁵ problem, adopting a date format that supports only a two-digit year is rather foolish. Consider the better date format shown in Figure 3-7.

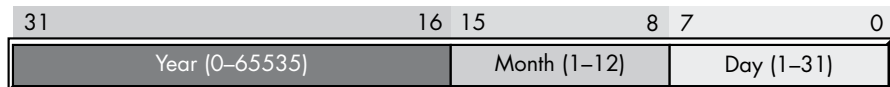


Figure 3-7: Long packed date format (32 bits)

Because there are more bits in a 32-bit variable than are needed to hold the date, even accounting for years in the range 0 through 65,535, this format allots a full byte for the month and day fields. An application can manipulate these two fields as byte objects, reducing the overhead to pack and unpack these fields on processors that support byte access. This leaves fewer bits for the year, but 65,536 years is probably sufficient (it's a safe bet that your software won't be in use 63,000 years from now).

You could argue that this is no longer a packed date format. After all, we needed three numeric values, two of which fit just nicely into 1 byte each and one that should have at least 2 bytes. This “packed” date format consumes the same 4 bytes as the unpacked version, not the fewest bits possible. So, in this example, packed effectively means *packaged* or *encapsulated*. By packing the data into a double-word variable, the program can treat the date value as a single data value rather than as three separate variables. This means that you can often get away with a single machine instruction to operate on this data rather than three separate instructions.

Another difference between this long packed date format and the short date format in Figure 3-6 is that this long date format rearranges the Year, Month, and Day fields. This allows you to easily compare two dates using an unsigned integer comparison. Consider the following HLA/assembly code:

```

mov( Date1, eax );          // Assume Date1 and Date2 are double-word variables
if( eax > Date2 ) then      // using the long packed date format.

    << do something if Date1 > Date2 >>

endif;

```

5. This was a software engineering near-disaster that occurred because programmers in the 1900s encoded dates using only two digits. They realized that when the year 2000 came along, programs wouldn't be able to differentiate 1900 and 2000.

Had you kept the different date fields in separate variables or organized the fields differently, you wouldn't have been able to compare `Date1` and `Date2` in such a straightforward way. Even if you don't realize any space savings, packing data can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

Some high-level languages provide built-in support for packed data. For example, in C you can define structures like the following:

```
struct
{
    unsigned bits0_3   :4;
    unsigned bits4_11  :8;
    unsigned bits12_15 :4;
    unsigned bits16_23 :8;
    unsigned bits24_31 :8;
} packedData;
```

This structure specifies that each field is an unsigned object that holds 4, 8, 4, 8, and 8 bits, respectively. The `:n` item after each declaration specifies the *minimum* number of bits the compiler will allocate for the given field.

Unfortunately, it isn't possible to show how a C/C++ compiler will allocate the values from a 32-bit double word among the fields, because C/C++ compiler implementers are free to implement these bit fields any way they see fit. The arrangement of the bits within the bit string is arbitrary (for example, the compiler could allocate the `bits0_3` field in bits 28 through 31 of the ultimate object). The compiler can also inject extra bits between fields, or use a larger number of bits for each field (which is actually the same thing as injecting extra padding bits between fields). Most C compilers attempt to minimize extraneous padding, but compilers (especially on different CPUs) do vary. Therefore, C/C++ struct bit field declarations are almost guaranteed to be nonportable, and you can't really count on what the compiler is going to do with those fields.

The advantage of using the compiler's built-in data-packing capabilities is that the compiler automatically packs and unpacks the data for you. Given the following C/C++ code, the compiler would automatically emit the necessary machine instructions to store and retrieve the individual bit fields for you:

```
struct
{
    unsigned year  :7;
    unsigned month :4;
    unsigned day   :5;
} ShortDate;

    . . .
    ShortDate.day = 28;
    ShortDate.month = 2;
    ShortDate.year = 3; // 2003
```

3.7 Packing and Unpacking Data

The advantage of packed data types is efficient memory use. Consider the Social Security number (SSN) used in the United States, a nine-digit identification code in the following form (each X represents a single decimal digit):

XXX-XX-XXXX

Encoding an SSN using three separate (32-bit) integers takes 12 bytes. That's more than the 11 bytes needed to represent the number using an array of characters. A better solution is to encode each field using short (16-bit) integers. Now it takes only 6 bytes to represent the SSN. Because the middle field in the SSN is always between 0 and 99, we can actually shave one more byte off the size of this structure by encoding the middle field with a single byte. Here's a sample Free Pascal/Delphi record structure that defines this data structure:

```
SSN :record

    FirstField: smallint; // smallints are 16 bits in Free Pascal/Delphi
    SecondField: byte;
    ThirdField: smallint;

end;
```

If we drop the hyphens in the SSN, the result is a nine-digit number. Because we can exactly represent all nine-digit values using 30 bits, we could encode any legal SSN using a 32-bit integer. However, some software that manipulates SSNs may need to operate on the individual fields. This means using expensive division, modulo, and multiplication operators in order to extract fields from a SSN you've encoded in a 32-bit integer format. Furthermore, converting SSNs to and from strings is more complicated when you're using the 32-bit format.

Conversely, it's easy to insert and extract individual bit fields using fast machine instructions, and it's also less work to create a standard string representation (including the hyphens) of one of these fields. Figure 3-8 shows a straightforward implementation of the SSN packed data type using a separate string of bits for each field (note that this format uses 31 bits and ignores the HO bit).

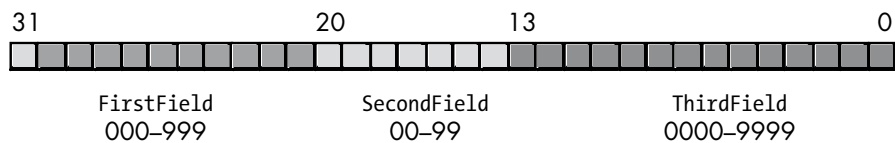


Figure 3-8: SSN packed fields encoding

Fields that begin at bit position 0 in a packed data object can be accessed most efficiently, so you should arrange the fields in your packed data type such that the field you access most often⁶ begins at bit 0. If you have no idea which field you'll access most often, assign the fields so they begin on a byte boundary. If there are unused bits in your packed type, spread them throughout the structure so that individual fields begin on a byte boundary and those fields consume multiples of 8 bits.

We've got only one unused bit in the SSN example shown in Figure 3-8, but it turns out that we can use this extra bit to align two fields on a byte boundary and ensure that one of those fields occupies a bit string whose length is a multiple of 8 bits. Consider Figure 3-9, which shows a rearranged version of our SSN data type.

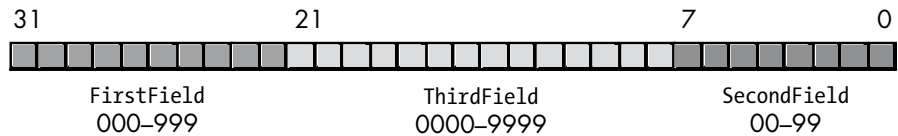


Figure 3-9: A (possibly) improved encoding of the SSN

One problem with the data format in Figure 3-9 is that we can't sort SSNs in an intuitive way by comparing 32-bit unsigned integers.⁷ If you intend to do a lot of sorting based on the entire SSN, the format in Figure 3-8 is probably better.

If this type of sorting isn't important to you, the format in Figure 3-9 has some advantages. This packed type actually uses 8 bits (rather than 7) to represent SecondField (along with moving SecondField down to bit position 0); the extra bit will always contain 0. This means that SecondField consumes bits 0 through 7 (a whole byte) and ThirdField begins on a byte boundary (bit position 8). ThirdField doesn't consume a multiple of 8 bits, and FirstField doesn't begin on a byte boundary, but we've done fairly well with this encoding, considering we only had one extra bit to play around with.

The next question is, "How do we access the fields of this packed type?" There are two separate activities here. We need to retrieve, or *extract*, the packed fields, and we need to *insert* data into these fields. The AND, OR, and SHIFT operations provide the tools for this.

When operating on these fields, it's convenient to work with three separate variables rather than with the packed data directly. For our SSN example, we can create the three variables—FirstField, SecondField, and ThirdField—and then extract the actual data from the packed value into these three variables, operate on the variables, and insert the data from the variables back into their fields when we're done.

6. Which field you'll access most often depends on the application.

7. "Intuitive" meaning that the first field is the most significant portion of the value, the second field is the next most significant, and the third field is the least significant component of the number.

To extract the `SecondField` data from the packed format shown in Figure 3-9 (remember, the field aligned to bit 0 is the easiest one to access), copy the data from the packed representation to the `SecondField` variable and then mask out all but the `SecondField` bits using the AND operation. Because `SecondField` is a 7-bit value, the mask is an integer containing 1s in bit positions 0 through 6 and 0s everywhere else. The following C/C++ code demonstrates how to extract this field into the `SecondField` variable (assuming *packedValue* is a variable holding the 32-bit packed SSN):

```
SecondField = packedValue & 0x7f;    // 0x7f = %0111_1111
```

Extracting fields that are not aligned at bit 0 takes a little more work. Consider the `ThirdField` entry in Figure 3-9. We can mask out all the bits associated with the first and second fields by logically ANDing the packed value with `%_11_1111_1111_1111_0000_0000` (`$3F_FF00`). However, this leaves the `ThirdField` value sitting in bits 8 through 21, which is not convenient for various arithmetic operations. The solution is to shift the masked value down 8 bits so that it's aligned at bit 0 in our working variable. The following Pascal/Delphi code does this:

```
ThirdField := (packedValue and $3fff00) shr 8;
```

You can also shift first and then do the logical AND operation (though this requires a different mask, `%11_1111_1111_1111` or `$3FFF`). Here's the C/C++/Swift code that extracts `ThirdField` using that technique:

```
ThirdField = (packedValue >> 8) & 0x3FFF;
```

To extract a field that is aligned against the HO bit, such as the first field in our SSN packed data type, shift the HO field down so that it's aligned at bit 0. The logical shift right operation automatically fills in the HO bits of the result with 0s, so no masking is necessary. The following Pascal/Delphi code demonstrates this:

```
FirstField := packedValue shr 22; // Delphi's SHR is a logical shift right.
```

In HLA/x86 assembly language, we can easily access data at any arbitrary byte boundary in memory. That allows us to treat both the second and third fields as though they are aligned at bit 0 in the data structure. In addition, because the `SecondField` value is an 8-bit value (with the HO bit always containing 0), it takes only a single machine instruction to unpack the data, as shown here:

```
movzx( (type byte packedValue), eax );
```

This instruction fetches the first byte of *packedValue* (which is the LO 8 bits of *packedValue* on the 80x86) and zero-extends this value to 32 bits in EAX (`movzx` stands for “move with zero extension”). The EAX register contains the `SecondField` value after this instruction executes.

The `ThirdField` value from our packed data type isn't an even multiple of 8 bits long, so we'll still need a masking operation to clear the unused bits from the 32-bit result we produce. However, because `ThirdField` is aligned on a byte (8-bit) boundary in our packed structure, we'll be able to avoid the shift operation that was necessary in the high-level code. Here's the HLA/x86 assembly code that extracts the third field from our *packedValue* object:

```
mov( (type word packedValue[1]), ax ); // Extracts bytes 1 & 2
                                         // from packedValue.
and( $3FFF, eax );                      // Clears all the undesired bits.
```

Extracting `FirstField` from the *packedValue* object in HLA/x86 assembly code is identical to the high-level code; we'll simply shift the upper 10 bits (which comprise `FirstField`) down to bit 0:

```
mov( packedValue, eax );
shr( 22, eax );
```

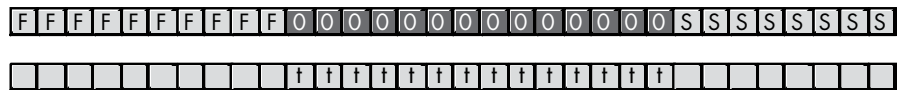
Assuming the data you want to insert appears in some variable and contains 0s in the unused bits, inserting a field into a packed object requires three operations. First, if necessary, you shift the field's data to the left so its alignment matches the corresponding field in the packed object. Next, clear the corresponding bits in the packed structure, then logically OR the shifted field into the packed object. Figure 3-10 shows the details of this operation.



Step 1: Align the bits in the `ThirdField` variable to bit position 8



Step 2: Mask out the corresponding bits in the packed structure



Step 3: Logically OR the two values to produce the final result



Final result

Figure 3-10: Inserting *ThirdField* into the *SSN* packed type

Here's the C/C++/Swift code that accomplishes the operation shown in Figure 3-10:

```
packedValue = (packedValue & 0xFFC000FF) | (ThirdField << 8 );
```

\$FFC000FF is the hexadecimal value that corresponds to 0s in bit positions 8 through 21 and 1s everywhere else.

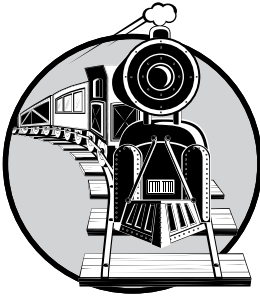
3.8 For More Information

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Boston: Addison-Wesley, 1998.

4

FLOATING-POINT REPRESENTATION



Floating-point arithmetic is an approximation of real arithmetic that solves the major problem with integer data types—the inability to represent fractional values.

However, the inaccuracies in this approximation can lead to serious defects in application software.

In order to write great software that produces correct results when using floating-point arithmetic, programmers must be aware of the machine's underlying numeric representation and exactly how floating-point arithmetic approximates real arithmetic.

4.1 Introduction to Floating-Point Arithmetic

There is an infinite number of possible real values. Floating-point representation uses a finite number of bits and, therefore, can represent a finite number of different values. When a given floating-point format cannot exactly represent some real value, the closest value that the format *can* exactly represent is

used. This section describes how the floating-point format works so you can better understand the drawbacks of these approximations.

Consider a couple of problems with integer and fixed-point formats. Integers cannot represent any fractional values, and they can represent only values in the range 0 through $2^n - 1$ or -2^{n-1} through $2^{n-1} - 1$. Fixed-point formats represent fractional values, but at the expense of the range of integer values they can represent. This problem, which the floating-point format solves, is one of *dynamic range*.

Consider a simple 16-bit unsigned fixed-point format that uses 8 bits for the fractional component and 8 bits for the integer component of the number. The integer component can represent values in the range 0 through 255, and the fractional component can represent the values 0 and fractions between 2^{-8} and 1 (with a resolution of about 2^{-8}). If in a string of calculations you need only 2 bits to represent the fractional values 0.0, 0.25, 0.5, and 0.75, the extra 6 bits in the fractional part of the number go to waste. Wouldn't it be nice if we could utilize those bits in the integer portion of the number to extend its range from 0 through 255 to 0 through 16,383? Well, that's the basic concept behind the floating-point representation.

In a floating-point value, the radix point (binary point) can float between digits in the number as needed. So, in a 16-bit binary number that needs only 2 bits of precision for the fractional component, the binary point can float down between bits 1 and 2, leaving bits 2 through 15 for the integer portion. A floating-point format needs one additional field to specify the position of the radix point within the number, equivalent to the exponent in scientific notation.

Most floating-point formats use some number of bits to represent a mantissa and a smaller number of bits to represent an exponent. The *mantissa* is a base value that usually falls within a limited range (for example, between 0 and 1). The *exponent* is a multiplier that, when applied to the mantissa, produces values outside this range. The big advantage of the mantissa/exponent configuration is that a floating-point format can represent values across a wide range. However, separating the number into these two parts means floating-point formats can represent only numbers with a specific number of *significant* digits. If the difference between the smallest and largest exponent is greater than the number of significant digits in the mantissa (and it usually is), then the floating-point format cannot exactly represent all the integers between the smallest and largest values in the floating-point representation.

To see the impact of limited-precision arithmetic, we'll adopt a simplified *decimal* floating-point format for our examples. Our floating-point format will use a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values, as shown in Figure 4-1.



Figure 4-1: Simple floating-point format

This particular floating-point representation can approximate all the values between 0.00 and 9.99×10^{99} . However, this format cannot represent all (integer) values in this range (that would take 100 digits of precision!). A value like 9,876,543,210 would be approximated with 9.88×10^9 (or $9.88\text{e}+9$ in programming language notation, which this book will generally use).

You cannot *exactly* represent as many different values with a floating-point format as with an integer format because the floating-point format encodes multiple representations (that is, different bit patterns) for the same value. In the simplified decimal floating-point format shown in Figure 4-1, for example, $1.00\text{e} + 1$ and $0.10\text{e} + 2$ are different representations of the same value. Because the number of different possible representations is finite, whenever a single value has two possible representations, that's one less unique value the format can represent.

Furthermore, the floating-point format, a form of scientific notation, complicates arithmetic somewhat. When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding $1.23\text{e}1$ and $4.56\text{e}0$, you could convert $4.56\text{e}0$ to $0.456\text{e}1$ and then add them. The result, $1.686\text{e}1$, does not fit into the three significant digits of our current format, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain $1.69\text{e}1$. The lack of *precision* (the number of digits or bits maintained in a computation) affects the *accuracy* (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating-point calculation were limited to three significant digits during computation, we would have had to truncate (throw away) the last digit of the smaller number, obtaining $1.68\text{e}1$, which is even less correct. Therefore, to improve the accuracy, we use extra digits during the calculation. These extra digits are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy lost during a single computation usually isn't bad. However, the error can accumulate over a sequence of floating-point operations and greatly affect the computation itself. For example, suppose we add $1.23\text{e}3$ and $1.00\text{e}0$. Adjusting the numbers so their exponents are the same before the addition produces $1.23\text{e}3 + 0.001\text{e}3$. The sum of these two values, even after rounding, is $1.23\text{e}3$. This might seem perfectly reasonable to you: if we can maintain only three significant digits, adding in a small value shouldn't affect the result. However, suppose we add $1.00\text{e}0$ to $1.23\text{e}3$ *10 times*. The first time we add $1.00\text{e}0$ to $1.23\text{e}3$, we get $1.23\text{e}3$. Likewise, we get this same result the second, third, fourth . . . and tenth time. Had we added $1.00\text{e}0$ to itself 10 times, then added the result ($1.00\text{e}1$) to $1.23\text{e}3$, we would obtain a different result, $1.24\text{e}3$. This is an important rule of limited-precision arithmetic:

The order of evaluation can affect the accuracy of the result.

Adding or subtracting numbers with relative magnitudes (that is, the sizes of the exponents) that are similar produces better results. If you're performing a chain calculation involving addition and subtraction, you

should group the operations so that you can add or subtract values whose magnitudes are close to one another before adding or subtracting values whose magnitudes are not as close.

Another problem with addition and subtraction is *false precision*. Consider the computation $1.23\text{e}0 - 1.22\text{e}0$. This produces $0.01\text{e}0$. Although this is mathematically equivalent to $1.00\text{e} - 2$, this latter form suggests that the last two digits (in the thousandths and ten-thousandths place) are both exactly 0. Unfortunately, we only have a single significant digit after this computation, which is in the hundredths place, and some FPUs or floating-point software packages might actually insert random digits (or bits) into the LO positions. This brings up a second important rule:

Whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating-point format.

Multiplication and division do not suffer from these problems, because you don't have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they exacerbate any accuracy error that already exists in a value. For example, if you multiply $1.23\text{e}0$ by 2, when you should be multiplying $1.24\text{e}0$ by 2, the result is even less accurate than it was. This brings up a third important rule:

When performing a chain of calculations involving addition, subtraction, multiplication, and division, perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiplication and division operations occur first. For example, suppose you want to compute the following:

$$x \times (y + z)$$

Normally, you would add y and z together and multiply their sum by x . However, you'll get a little more accuracy if you first transform the expression to the following:

$$x \times y + x \times z$$

Now you can compute the result by performing the multiplications first.¹

Multiplication and division have other problems as well. When you multiply two very large or very small numbers, *overflow* or *underflow* may occur. The same situation occurs when you divide a small number by a large number, or a large number by a small number. This brings up a fourth rule:

When multiplying and dividing sets of numbers, try to multiply and divide numbers that have the same relative magnitudes.

1. Of course, the drawback is that you must now perform two multiplications rather than one, so the result may be slower.

Comparing floating-point numbers is very dangerous. Given the inaccuracies inherent in any computation (including converting an input string to a floating-point value), you should *never* compare two floating-point values to see if they are equal. Different computations that produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0 and 1.69e0 should produce 3.00e0. Likewise, adding 1.50e0 and 1.50e0 should produce 3.00e0. However, were you to compare (1.31e0 + 1.69e0) to (1.50e0 + 1.50e0), you might find that these sums are *not* equal. Because two seemingly equivalent floating-point computations will not necessarily produce exactly equal results, a straight comparison for equality—which succeeds if and only if all bits (or digits) in the two operands are the same—may fail.

To test for equality between floating-point numbers, determine how much error (or tolerance) you'll allow in a comparison, and then check to see if one value is within this error range of the other, like so:

```
if( (Value1 >= (Value2 - error)) and (Value1 <= (Value2 + error)) then . . .
```

More efficient is to use a statement of the form:

```
if( abs(Value1 - Value2) <= error ) then . . .
```

The value for *error* should be slightly greater than the largest amount of error that will creep into your computations. The exact value depends upon the particular floating-point format you use and the magnitudes of the values you are comparing. So, the final rule is this:

When comparing two floating-point numbers for equality, always compare the values to see if the difference between two values is less than some small error value.

Checking two floating-point numbers for equality is a very famous problem, one that almost every introductory programming text discusses. The same problems with comparing for less than or greater than, however, are not as well known. Suppose that a sequence of floating-point calculations produces a result that is accurate only to within $\pm error$, even though the floating-point representation provides better accuracy than *error* suggests. If you compare such a result against some other calculation computed with less accumulated error, and those two values are very close to each other, then comparing them for less than or greater than may produce incorrect results.

For example, suppose that some chain of calculations in our simplified decimal representation produces 1.25, which is accurate only to ± 0.05 (that is, the real value could be somewhere between 1.20 and 1.30), and a second chain of calculations produces 1.27, which is accurate to the full precision of our floating-point representation (that is, the actual value, before rounding, is somewhere between 1.265 and 1.275). Comparing the result of the first calculation (1.25) to the result of the second calculation (1.27) finds that the first result is less than the second. Unfortunately, given the inaccuracy of the

first calculation, this might not be true—for example, if the correct result of the first computation is in the range 1.27 to 1.30 (exclusive).

About the only reasonable test is to see if the two values are within the error tolerance of each other. If so, treat the values as equal (neither is considered less than or greater than the other). If the values are not equal within the desired error tolerance, you can compare them to see if one value is less than or greater than the other. This is known as a *miserly approach*; that is, we try to find as few values that are less than or greater than as possible.

The other possibility is to use an *eager approach*, which attempts to make the result of the comparison true as often as possible. Given two values to compare and an error tolerance, here's how you'd eagerly compare the two values for less than or greater than:

```
if( A < (B + error) ) then Eager_A_lessthan_B;  
if( A > (B - error) ) then Eager_A_greaterthan_B;
```

Don't forget that calculations like $(B + \text{error})$ are subject to their own inaccuracies, depending on the relative magnitudes of the values B and error , and the inaccuracy of this calculation may affect the final result of the comparison.

NOTE

Due to space limitations, this book merely touches on some major problems that can occur when you're using floating-point values and why you can't treat floating-point arithmetic like real arithmetic. For further details, consult a good text on numerical analysis or even scientific computing. If you're going to be working with floating-point arithmetic, in any language, take some time to study the effects of limited-precision arithmetic on your computations.

4.2 IEEE Floating-Point Formats

When Intel planned to introduce a floating-point unit (FPU) for its original 8086 microprocessor, the company was smart enough to realize that the electrical engineers and solid-state physicists who design chips probably didn't have the necessary numerical analysis background to design a good floating-point representation. So, Intel went out and hired the best numerical analyst it could find to design a floating-point format for its 8087 FPU. That person then hired two other experts in the field, and the three of them (Kahan, Coonen, and Stone) designed the *KCS Floating-Point Standard*. They did such a good job that the IEEE organization used this format as the basis for the IEEE Std 754 floating-point format.

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating-point formats: single precision, double precision, and extended precision. The single- and double-precision formats corresponded to C's float and double types or FORTRAN's real and double precision types. Extended precision contains 16 extra bits that long chains of computations can use as guard bits before rounding down to a double-precision value when storing the result.

4.2.1 Single-Precision Floating-Point Format

The single-precision format uses a 24-bit mantissa and an 8-bit exponent. The mantissa represents a value between 1.0 and just less than 2.0. The HO bit of the mantissa is always 1 and represents a value just to the left of the binary point. The remaining 23 mantissa bits appear to the right of the binary point and represent the value:

1.mmmmmmm mmmmmmm mmmmmmm

The mantissa is always greater than or equal to 1 because of the implied 1 bit. Even if the other mantissa bits are all 0, the implied 1 bit always gives us the value 1. Each position to the right of the binary point represents a value (0 or 1) times a successive negative power of 2, but even if we had an almost infinite number of 1 bits after the binary point, they still would not add up to 2. So, the mantissa can represent values in the range 1.0 to just less than 2.0.

Some examples would probably be useful here. Consider the decimal value 1.7997. Here are the steps we could go through to compute the binary mantissa for this value:

1. Subtract 2^0 from 1.7997 to produce 0.7997 and
%1.000000000000000000000000.
2. Subtract 2^{-1} ($1/2$) from 0.7997 to produce 0.2997 and
%1.100000000000000000000000.
3. Subtract 2^{-2} ($1/4$) from 0.2997 to produce 0.0497 and
%1.110000000000000000000000.
4. Subtract 2^{-5} ($1/32$) from 0.0497 to produce 0.0185 and
%1.110010000000000000000000.
5. Subtract 2^{-6} ($1/64$) from 0.0185 to produce 0.00284 and
%1.110011000000000000000000.
6. Subtract 2^{-9} ($1/512$) from 0.00284 to produce 0.000871 and
%1.110011001000000000000000.
7. Subtract 2^{-10} ($1/1,024$) from 0.000871 to (approximately) produce 0 and
%1.110011001100000000000000.

Although there is an infinite number of values between 1 and 2, we can represent only 8 million (2^{23}) of them because we use a 23-bit mantissa (the 24th bit is always 1), and therefore have only 23 bits of precision.

The mantissa uses a *one's complement* format rather than two's complement. This means that the 24-bit value of the mantissa is simply an unsigned binary number, and the sign bit, in bit position 31, determines whether that value is positive or negative. One's complement has the unusual property that there are two representations for 0 (with the sign bit set or clear). Generally, this is important only to the person designing the floating-point software or hardware system. We'll assume that the value 0 always has the sign bit clear.

The single-precision floating-point format is shown in Figure 4-2.

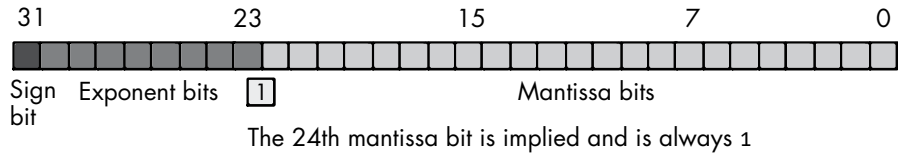


Figure 4-2: Single-precision (32-bit) floating-point format

We represent values outside the range of the mantissa by raising 2 to the power specified by the exponent and then multiplying the result by the mantissa. The exponent is 8 bits and uses an *excess-127* format (sometimes called *bias-127 exponents*). In excess-127 format, the exponent 2^0 is represented by the value 127 (\$7f). To convert an exponent to excess-127 format, add 127 to the exponent value. For example, the single-precision representation for 1.0 is \$3f800000. The mantissa is 1.0 (including the implied bit) and the exponent is 2^0 , encoded as 127 (\$7f). The representation for 2.0 is \$40000000, with the exponent 2^1 encoded as 128 (\$80).

The excess-127 exponent makes it easy to compare two floating-point numbers for less than or greater than as though they were unsigned integers, as long as we handle the sign bit (bit 31) separately. If the signs of the two values are not equal, then the positive value (the one with bit 31 set to 0) is greater than the value that has the HO bit set to 1.² If the sign bits are both 0, we use a straight unsigned binary comparison. If the signs are both 1, we do an unsigned comparison but invert the result (that is, we treat less than as greater than and vice versa). On some CPUs, where a 32-bit unsigned comparison is much faster than a 32-bit floating-point comparison, it's probably worthwhile to do the comparison using integer arithmetic rather than floating-point arithmetic.

A 24-bit mantissa provides approximately $6\frac{1}{2}$ decimal digits of precision (one-half digit of precision means that the first six digits can be in the range 0..9, but the seventh digit can only be in the range 0 through x where $x < 9$ and is generally close to 5). With an 8-bit excess-127 exponent, the dynamic range of single-precision floating-point numbers is approximately $2^{\pm 128}$ or about $10^{\pm 38}$.

Although single-precision floating-point numbers are perfectly suitable for many applications, the dynamic range is unsuitable for many financial, scientific, and other applications. Furthermore, during long chains of computations, the limited accuracy may introduce significant error. For serious calculations, we need a floating-point format with more precision.

2. Actually, there are a couple of exceptions. The floating-point format has two representations for 0—one with the sign bit set and one with the sign bit clear; a floating-point comparison should treat these two values as equal. Likewise, there are a couple of special floating-point values that are incomparable, and the comparison operation must consider those values as well.

4.2.2 Double-Precision Floating-Point Format

The double-precision format helps overcome the problems of the single-precision floating-point. Using twice the space, the double-precision format has an 11-bit excess-1,023 exponent, a 53-bit mantissa (including an implied HO bit of 1), and a sign bit. This provides a dynamic range of about $10^{\pm 308}$ and 15 to 16+ digits of precision, which is sufficient for most applications. Double-precision floating-point values take the form shown in Figure 4-3.

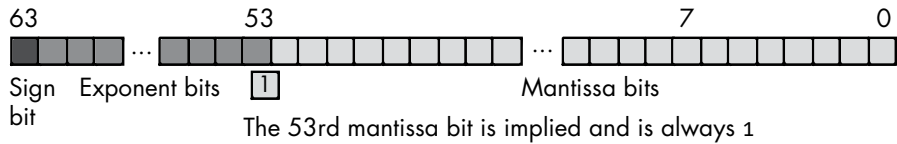


Figure 4-3: Double-precision (64-bit) floating-point format

4.2.3 Extended-Precision Floating-Point Format

To ensure accuracy during long chains of computations involving double-precision floating-point numbers, Intel designed the extended-precision format. The extended-precision format uses 80 bits: a 64-bit mantissa, a 15-bit excess-16,383 exponent, and a 1-bit sign. The mantissa does not have an implied HO bit that is always 1. The format for the extended-precision floating-point value appears in Figure 4-4.

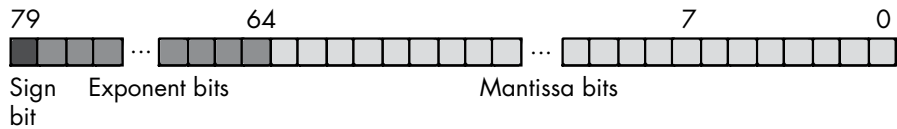


Figure 4-4: Extended-precision (80-bit) floating-point format

On the 80x86 FPUs, all computations use the extended-precision form. Whenever you load a single- or double-precision value, the FPU automatically converts it to an extended-precision value. Likewise, when you store a single- or double-precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. The extended-precision format guarantees the inclusion of a large number of guard bits in 32- and 64-bit computations, which helps ensure (but not guarantee) that you'll get full 32- or 64-bit accuracy in your computations. Some error will inevitably creep into the LO bits because the FPUs provide no guard bits for 80-bit computations (the FPU uses only 64 mantissa bits during 80-bit computations). While you can't assume that you'll get an accurate 80-bit computation, you can usually do better than 64 bits when using the extended-precision format.

Non-Intel CPUs that support floating-point arithmetic generally provide only the 32-bit and 64-bit formats. Therefore, calculations on those CPUs may produce less accurate results than the equivalent string of calculations on the 80x86 using 80-bit calculations. Also note that modern x86-64 CPUs have additional floating-point hardware as part of the SSE

extensions; however, those SSE extensions support only 64- and 32-bit floating-point calculations.

4.2.4 Quad-Precision Floating-Point Format

The original 80-bit extended-precision floating-point format was a stopgap measure. From a “types should be consistent” point of view, the proper extension to the 64-bit floating-point format should have been a 128-bit floating-point format. Alas, when Intel was working on floating-point formats in the late 1970s, a quad-precision (128-bit) floating-point format was too expensive to implement in hardware, so the 80-bit extended-precision format became the interim compromise. Today, a few CPUs (such as IBM’s POWER9 and later-version ARMs) are capable of quad-precision floating-point arithmetic.

The IEEE Std 754 quad-precision floating-point format uses a single sign bit, a 15-bit excess-16,383 biased exponent, and a 112-bit (with implied 113th bit) mantissa (see Figure 4-5). This provides 36 decimal digits of precision and exponents in the approximate range $10^{\pm 4932}$.

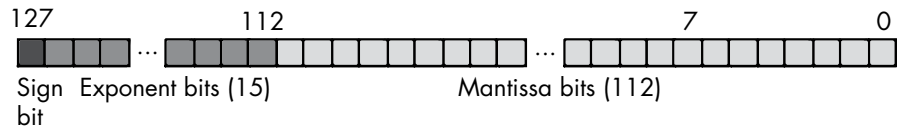


Figure 4-5: Extended-precision (80-bit) floating-point format

4.3 Normalization and Denormalized Values

To maintain maximum precision during floating-point computations, most computations use *normalized* values. A normalized floating-point value is one whose HO mantissa bit contains 1. A floating-point computation will be more accurate if it involves only normalized values because the mantissa has that many fewer bits of precision available for computation if several HO bits of the mantissa are all 0.

You can normalize almost any unnormalized value by shifting the mantissa bits to the left and decrementing the exponent until a 1 appears in the mantissa’s HO bit.³ Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating-point value by 2. Likewise, whenever you decrement the exponent, you divide the floating-point value by 2. By the same token, shifting the mantissa to the left one bit position multiplies the floating-point value by 2, and shifting it to the right divides the floating-point value by 2. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value

3. In the rare case where you wind up with more than one bit to the left of the binary point, you can normalize the mantissa by shifting its bits to the right one position and incrementing the exponent.

of the floating-point number (this is why, as you saw earlier, there are multiple representations for certain numbers in the floating-point format).

Here's an example of an unnormalized value:

$$0.100000 \times 2^1$$

Shift the mantissa to the left one position and decrement the exponent to normalize it:

$$1.000000 \times 2^0$$

There are two important cases in which a floating-point number cannot be normalized. First, 0 cannot be normalized because the floating-point representation contains all 0 bits in the exponent and mantissa fields. This, however, is not a problem, because we can exactly represent 0 with a single 0 bit, and extra bits of precision are unnecessary.

We also cannot normalize a floating-point number when we have some HO bits in the mantissa that are 0 but the biased exponent⁴ is also 0 (and we can't decrement it to normalize the mantissa). Rather than prohibiting certain small values whose HO mantissa bits and biased exponent are 0 (the most negative exponent possible), the IEEE standard permits special *denormalized* values in these cases.⁵ Although the use of denormalized values enables IEEE floating-point computations to produce better results than if underflow occurred, denormalized values offer fewer bits of precision.

4.4 Rounding

During a calculation, floating-point arithmetic functions may produce a result with greater precision than the floating-point format supports (the *guard bits* in the calculation maintain this extra precision). When the calculation is complete and the code needs to store the result back into a floating-point variable, something must be done about those extra bits of precision. How the system uses guard bits to affect the remaining bits is known as *rounding*, and how rounding is done can affect the accuracy of the computation. Traditionally, floating-point software and hardware use one of four different ways to round values: truncation, rounding up, rounding down, or rounding to nearest.

Truncation is easy, but it generates the least accurate results in a chain of computations. Few modern floating-point systems use truncation except as a means for converting floating-point values to integers (truncation is the standard conversion for coercing a floating-point value to an integer).

4. "Biased" means to add an offset to the value—for example, an excess-127 exponent has a bias of 127.

5. The alternative would be to underflow the values to 0.

Rounding up leaves the value alone if the guard bits are all 0, but if the current mantissa does not exactly fit into the destination bits, then rounding up sets the mantissa to the smallest possible larger value in the floating-point format. Like truncation, this is not a normal rounding mode. It is, however, useful for implementing functions like `ceil()`, which rounds a floating-point value to the smallest possible larger integer.

Rounding down is just like rounding up, except it rounds the result to the largest possible smaller value. This may sound like truncation, but there's a subtle difference: truncation always rounds toward 0. For positive numbers, truncation and rounding down do the same thing. For negative values, truncation simply uses the existing bits in the mantissa, whereas rounding down will add a 1 bit to the LO position if the result was negative. This is also not a normal rounding mode, but it's useful for implementing functions like `floor()`, which rounds a floating-point value to the largest possible smaller integer.

Rounding to nearest is the most intuitive way to process the guard bits. If the value of the guard bits is less than half the value of the mantissa's LO bit, then rounding to nearest truncates the result to the largest possible smaller value (ignoring the sign). If the guard bits represent some value that is greater than half of the value of the LO mantissa bit, then rounding to nearest rounds the mantissa to the smallest possible greater value (ignoring the sign). If the guard bits represent a value that is exactly half the value of the mantissa's LO bit, then the IEEE floating-point standard says that half the time it should round up and half the time it should round down. You do this by rounding the mantissa to the value that has a 0 in the LO bit position. That is, if the current mantissa already has a 0 in its LO bit, you use the current mantissa; if the current mantissa has a 1 in its LO bit, then you add 1 to round it up to the smallest possible larger value with a 0 in the LO bit. This scheme, mandated by the IEEE floating-point standard, produces the best possible result when loss of precision occurs.

Here are some examples of rounding, using 24-bit mantissas, with 4 guard bits (that is, these examples round 28-bit numbers to 24-bit numbers using the rounding to nearest algorithm):

```

1.000_0100_1010_0100_1001_0101_0001 -> 1.000_0100_1010_0100_1001_0101
1.000_0100_1010_0100_1001_0101_1100 -> 1.000_0100_1010_0100_1001_0110
1.000_0100_1010_0100_1001_0101_1000 -> 1.000_0100_1010_0100_1001_0110

1.000_0100_1010_0100_1001_0100_0001 -> 1.000_0100_1010_0100_1001_0100
1.000_0100_1010_0100_1001_0100_1100 -> 1.000_0100_1010_0100_1001_0101
1.000_0100_1010_0100_1001_0100_1000 -> 1.000_0100_1010_0100_1001_0100

```

4.5 Special Floating-Point Values

The IEEE floating-point format provides a special encoding for several special values. In this section, we'll look these special values, their purpose and meaning, and their representation in the floating-point format.

Under normal circumstances, the exponent bits of a floating-point number do not contain all 0s or all 1s. An exponent containing all 1 or 0 bits indicates a special value.

If the exponent contains all 1s and the mantissa is nonzero (discounting the implied bit), then the HO bit of the mantissa (again discounting the implied bit) determines whether the value represents a *quiet not-a-number* (QNaN) or a *signaling not-a-number* (SNaN) (see Table 4-1). These not-a-number (NaN) results tell the system that some serious miscalculation has taken place and that the result of the calculation is completely undefined. QNaNs represent *indeterminate* results, while SNaNs specify that an *invalid* operation has taken place. Any calculation involving a NaN produces a NaN result, regardless of the values of any other operand(s). Note that the sign bit is irrelevant for NaNs. The binary representations of NaNs are shown in Table 4-1.

Table 4-1: Binary Representations for NaN

NaN	FP format	Value
SNaN	32 bits	%s_11111111_0xxxx...xx (The value of s is irrelevant—at least one of the x bits must be nonzero.)
SNaN	64 bits	%s_1111111111_0xxxxx...x (The value of s is irrelevant—at least one of the x bits must be nonzero.)
SNaN	80 bits	%s_1111111111_0xxxxx...x (The value of s is irrelevant—at least one of the x bits must be nonzero.)
QNaN	32 bits	%s_11111111_1xxxx...xx (The value of s is irrelevant.)
QNaN	64 bits	%s_1111111111_1xxxxx...x (The value of s is irrelevant.)
QNaN	80 bits	%s_1111111111_1xxxxx...x (The value of s is irrelevant.)

Two other special values are represented when the exponent contains all 1 bits, and the mantissa contains all 0s. In such a case, the sign bit determines whether the result is the representation for *+infinity* or *-infinity*. Whenever a calculation involves infinity as one of the operands, the result will be one of the (well-defined) values found in Table 4-2.

Table 4-2: Operations Involving Infinity

Operation	Result
$n / \pm\text{infinity}$	0
$\pm\text{infinity} \times \pm\text{infinity}$	$\pm\text{infinity}$
$\pm\text{nonzero} / 0$	$\pm\text{infinity}$
$\text{infinity} + \text{infinity}$	infinity
$n + \text{infinity}$	infinity
$n - \text{infinity}$	-infinity
$\pm 0 / \pm 0$	NaN
$\text{infinity} - \text{infinity}$	NaN
$\pm\text{infinity} / \pm\text{infinity}$	NaN
$\pm\text{infinity} \times 0$	NaN

Finally, if the exponent bits are all 0, the sign bit indicates which of the two special values, -0 or $+0$, the floating-point number represents. Because the floating-point format uses a one's complement notation, there are two separate representations for 0. Note that with respect to comparisons, arithmetic, and other operations, $+0$ is equal to -0 .

USING MULTIPLE REPRESENTATIONS OF ZERO

The IEEE floating-point format supports both $+0$ and -0 (depending on the value of the sign bit), which are treated as equivalent by arithmetic calculations and comparisons—the sign bit is ignored. Software operating on floating-point values that represent 0 can use the sign bit as a flag to indicate different things. For example, you could use the sign bit to indicate that the value is exactly 0 (with the sign bit clear) or to indicate that it is nonzero but too small to represent with the current format (with the sign bit set). Intel recommends using the sign bit to indicate that 0 was produced via underflow of a negative value (with the sign bit set) or underflow of a positive number (with the sign bit clear). Presumably, their FPUs set the sign bit according to their recommendations when the FPUs produce a 0 result.

4.6 Floating-Point Exceptions

The IEEE floating-point standard defines certain degenerate conditions under which the floating-point processor (or software-implemented floating-point code) should notify the application software. These exceptional conditions include the following:

- Invalid operation
- Division by zero

- Denormalized operand
- Numeric overflow
- Numeric underflow
- Inexact result

Of these, inexact result is the least serious, because most floating-point calculations will produce an inexact result. A denormalized operand also isn't too serious (though this exception indicates that your calculation may be less accurate as a result of less available precision). The other exceptions indicate a more serious problem, and you shouldn't ignore them.

How the computer system notifies your application of these exceptions depends on the CPU/FPU, operating system, and programming language, so we can't really go into how you might handle these exceptions. Generally, though, you can use the exception handling facilities in your programming language to trap these conditions as they occur. Note that most computer systems won't notify you when one of the exceptional conditions exists unless you explicitly set up a notification.

4.7 Floating-Point Operations

Although most modern CPUs support an FPU that does floating-point arithmetic in hardware, it's worthwhile to develop a set of software floating-point arithmetic routines to get a solid feel for what's involved. Generally, you'd use assembly language to write the math functions because speed is a primary design goal for a floating-point package. However, because here we're writing a floating-point package simply to get a clearer picture of the process, we'll opt for code that is easy to write, read, and understand.

As it turns out, floating-point addition and subtraction are easy to do in a high-level language like C/C++ or Pascal, so we'll implement these functions in these languages. Floating-point multiplication and division are easier to do in assembly language than in a high-level language, so we'll write those routines using High-Level Assembly (HLA).

4.7.1 Floating-Point Representation

This section will use the IEEE 32-bit single-precision floating-point format (shown earlier in Figure 4-2), which uses a one's complement representation for signed values. This means that the sign bit (bit 31) contains a 1 if the number is negative and a 0 if the number is positive. The exponent is an 8-bit excess-127 exponent sitting in bits 23 through 30, and the mantissa is a 24-bit value with an implied HO bit of 1. Because of the implied HO bit, this format does not support denormalized values.

4.7.2 Floating-Point Addition and Subtraction

Addition and subtraction use essentially the same code. After all, computing $X - Y$ is equivalent to computing $X + (-Y)$. If we can add a negative number to some other value, then we can also perform subtraction by first

negating some number and then adding it to another value. And because the IEEE floating-point format uses the one's complement representation, negating a value is trivial—we just invert the sign bit.

Because we're using the standard IEEE 32-bit single-precision floating-point format, we could theoretically get away with using the C/C++ `float` data type (assuming the underlying C/C++ compiler also uses this format, as most do on modern machines). However, you'll soon see that when doing floating-point calculations in software, we need to manipulate various fields within the floating-point format as bit strings and integer values. Therefore, it's more convenient to use a 32-bit unsigned integer type to hold the bit representation for our floating-point values. To avoid confusing our real values with actual integer values in a program, we'll define the following real data type, which assumes that unsigned longs are 32-bit values in your implementation of C/C++ (this section assumes the `uint32_t` type achieves that, which is something like `typedef unsigned long uint32_t`), and declare all our real variables using this type:

```
typedef uint32_t  real;
```

One advantage of using the same floating-point format that C/C++ uses for `float` values is that we can assign floating-point literal constants to our real variables, and we can perform other floating-point operations such as input and output using existing library routines. However, one potential problem is that C/C++ will attempt to automatically convert between integer and floating-point formats if we use a real variable in a floating-point expression (remember, as far as C/C++ is concerned, `real` is just an unsigned long integer value). This means that we need to tell the compiler to treat the bit patterns found in our real variables as though they were `float` objects.

A simple type coercion like `(float) realVariable` won't work. The C/C++ compiler will emit code to convert the integer it believes `realVariable` contains into the equivalent floating-point value. However, we want the C/C++ compiler to treat the bit pattern it finds in `realVariable` as a `float` without doing any conversion. The following C/C++ macro is a sneaky way to do this:

```
#define asreal(x) (*((float *) &x))
```

This macro requires a single parameter that must be a real variable. The result is a variable that the compiler believes is a `float` variable.

Now that we have our `float` variable, we'll develop two C/C++ functions to compute floating-point addition and subtraction: `fpadd()` and `fpsub()`. These two functions each take three parameters: the left and right operands of the operator and a pointer to a destination where these functions will store their result. The prototypes for these functions are the following:

```
void fpadd( real left, real right, real *dest );  
void fpsub( real left, real right, real *dest );
```

The `fpsub()` function negates the right operand and calls the `fpadd()` function. Here's the code for the `fpsub()` function:

```
void fpsub( real left, real right, real *dest )
{
    right = right ^ 0x80000000; // Invert the sign bit of the right operand.
    fpadd( left, right, dest ); // Let fpadd do the real work.
}
```

The `fpadd()` function is where all the real work is done. To make `fpadd()` a little easier to understand and maintain, we'll decompose it into several different functions that help with various tasks. In an actual software floating-point library routine, you wouldn't do this decomposition, because the extra subroutine calls would be a little slower; however, we're developing `fpadd()` for educational purposes, and besides, if you need high-performance floating-point addition, you'll probably use a hardware FPU rather than a software implementation.

The IEEE floating-point formats are good examples of packed data types. As you've seen in previous chapters, packed data types are great for reducing storage requirements for a data type, but not so much when you need to use the packed fields in actual calculations. Therefore, one of the first things our floating-point functions will do is unpack the sign, exponent, and mantissa fields from the floating-point representation.

The first unpacking function, `extractSign()`, extracts the sign bit (bit 31) from our packed floating-point representation and returns the value 0 (for positive numbers) or 1 (for negative numbers).

```
inline int extractSign( real from )
{
    return( from >> 31);
}
```

This code could have also extracted the sign bit using this (possibly more efficient) expression:

```
(from & 0x80000000) != 0
```

However, shifting bit 31 down to bit 0 is, arguably, easier to understand.

The next utility function, `extractExponent()`, unpacks the exponent from bits 23 through 30 in the packed real format. It does this by shifting the real value to the right by 23 bits, masking out the sign bit, and converting the excess-127 exponent to a two's complement format (by subtracting 127).

```
inline int extractExponent( real from )
{
    return ((from >> 23) & 0xff) - 127;
}
```

Next is the `extractMantissa()` function, which extracts the mantissa from the real value. To extract the mantissa, we must mask out the exponent and sign bits and then insert the implied HO bit of 1. The only catch is that we must return 0 if the entire value is 0.

```
inline int extractMantissa( real from )
{
    if( (from & 0x7fffffff) == 0 ) return 0;
    return ((from & 0x7FFFFF) | 0x800000 );
}
```

As you learned earlier, whenever adding or subtracting two values using scientific notation (which the IEEE floating-point format uses), you must first adjust the two values so that they have the same exponent. For example, to add the two decimal (base-10) numbers $1.2345e3$ and $8.7654e1$, we must first adjust one or the other so that their exponents are the same. We can reduce the exponent of the first number by shifting the decimal point to the right. For example, the following values are all equivalent to $1.2345e3$:

$12.345e2$ $123.45e1$ 1234.5 $12345e-1$

Likewise, we can increase the value of an exponent by shifting the decimal point to the left. The following values are all equal to $8.7654e1$:

$0.87654e2$ $0.087654e3$ $0.0087654e4$

For floating-point addition and subtraction involving binary numbers, we can make the binary exponents the same by shifting the mantissa one position to the left and decrementing the exponent, or by shifting the mantissa one position to the right and incrementing the exponent.

Shifting the mantissa bits to the right means that we reduce the precision of our number (because the bits wind up going off the LO end of the mantissa). To maintain as much accuracy as possible in our calculations, we shouldn't truncate the bits we shift out of the mantissa, but rather round the result to the nearest value we can represent with the remaining mantissa bits. These are the IEEE rules for rounding, in order:

1. Truncate the result if the last bit shifted out was a 0.
2. Increment the mantissa by 1 if the last bit shifted out was a 1 and there was at least one bit set to 1 in all the other bits that were shifted out.⁶
3. If the last bit we shifted out was a 1, and all the other bits were 0s, then round the resulting mantissa up by 1 if the mantissa's LO bit contains a 1.

Shifting the mantissa and rounding it is a relatively complex operation, and it will occur a couple of times in the floating-point addition code.

6. If the algorithm shifts out only a single bit, you assume that "all the other bits" are 0s.

Therefore, it's another candidate for a utility function. Here's the C/C++ code that implements this function, `shiftAndRound()`:

```
void shiftAndRound( uint32_t *valToShift, int bitsToShift )
{
    // Masks is used to mask out bits to check for a "sticky" bit.
    static unsigned masks[24] =
    {
        0, 1, 3, 7, 0xf, 0x1f, 0x3f, 0x7f,
        0xff, 0x1ff, 0x3ff, 0x7ff, 0xfff, 0x1fff, 0x3fff, 0x7fff,
        0xffff, 0x1ffff, 0x3ffff, 0x7ffff, 0xfffff, 0x1fffff, 0x3fffff,
        0x7fffff
    };

    // H0masks: Masks out the H0 bit of the value masked by the masks entry.
    static unsigned H0masks[24] =
    {
        0,
        1, 2, 4, 0x8, 0x10, 0x20, 0x40, 0x80,
        0x100, 0x200, 0x400, 0x800, 0x1000, 0x2000, 0x4000, 0x8000,
        0x10000, 0x20000, 0x40000, 0x80000, 0x100000, 0x200000, 0x400000
    };

    // shiftedOut: Holds the value that will be shifted out of a mantissa
    // during the denormalization operation (used to round a denormalized
    // value).
    int shiftedOut;

    assert( bitsToShift <= 23 );

    // Okay, first grab the bits we're going to shift out (so we can determine
    // how to round this value after the shift).
    shiftedOut = *valToShift & masks[ bitsToShift ];

    // Shift the value to the right the specified number of bits.
    // Note: bit 31 is always 0, so it doesn't matter if the C
    // compiler does a logical shift right or an arithmetic shift right.
    *valToShift = *valToShift >> bitsToShift;

    // If necessary, round the value:

    if( shiftedOut > H0masks[ bitsToShift ] )
    {
        // If the bits we shifted out are greater than 1/2 the L0 bit, then
        // round the value up by 1.

        *valToShift = *valToShift + 1;
    }
    else if( shiftedOut == H0masks[ bitsToShift ] )
    {
        // If the bits we shifted out are exactly 1/2 of the L0 bit's value,
        // then round the value to the nearest number whose L0 bit is 0.

        *valToShift = *valToShift + (*valToShift & 1);
    }
}
```

```

    }
    // else
    // We round the value down to the previous value. The current
    // value is already truncated (rounded down), so we don't have to do
    // anything.
}

```

The “trick” in this code is that it uses a couple of lookup tables, masks and H0masks, to extract those bits that the mantissa will use from the shift right operation. The masks table entries contain 1 bits (set bits) in the positions that will be lost during the shift. The H0masks table entries contain a single set bit in the position specified by the index into the table; that is, the entry at index 0 contains a 1 in bit position 0, the entry at index 1 contains a 1 in bit position 1, and so on. This code selects an entry from each of these tables based on the number of mantissa bits it needs to shift to the right.

If the original mantissa value, logically ANDed with the appropriate entry in masks, is greater than the corresponding entry in H0masks, then the shiftAndRound() function rounds the shifted mantissa to the next greater value. If the ANDed mantissa value is equal to the corresponding H0masks element, this code rounds the shifted mantissa value according to its LO bit (note that the expression (*valToShift & 1) produces 1 if the mantissa’s LO bit is 1, and it produces 0 otherwise). Finally, if the ANDed mantissa value is less than the entry from the H0masks table, then this code doesn’t have to do anything because the mantissa is already rounded down.

Once we’ve adjusted one of the values so that the exponents of both operands are the same, the next step in the addition algorithm is to compare the signs of the values. If the signs of the two operands are the same, we add their mantissas (using a standard integer add operation). If the signs differ, we have to subtract, rather than add, the mantissas. Because floating-point values use one’s complement representation, and standard integer arithmetic uses two’s complement, we cannot simply subtract the negative value from the positive value. Instead, we have to subtract the smaller value from the larger value and determine the sign of the result based on the signs and magnitudes of the original operands. Table 4-3 describes how to accomplish this.

Table 4-3: Dealing with Operands That Have Different Signs

Left sign	Right sign	Left mantissa > right mantissa?	Compute mantissa as	Result sign is
–	+	Yes	<i>LeftMantissa - RightMantissa</i>	–
+	–	Yes	<i>LeftMantissa - RightMantissa</i>	+
–	+	No	<i>RightMantissa - LeftMantissa</i>	+
+	–	No	<i>RightMantissa - LeftMantissa</i>	–

Whenever you’re adding or subtracting two 24-bit numbers, it’s possible to produce a result that requires 25 bits (in fact, this is common when you’re dealing with normalized values). Immediately after an addition or

subtraction, the floating-point code has to check the result to see if overflow has occurred. If so, it needs to shift the mantissa right by 1 bit, round the result, and then increment the exponent. After completing this step, all that remains is to pack the resulting sign, exponent, and mantissa fields into the 32-bit IEEE floating-point format. The following `packFP()` function is responsible for packing the sign, exponent, and mantissa fields into the 32-bit floating-point format:

```
inline real packFP( int sign, int exponent, int mantissa )
{
    return
        (real)
        (
            (sign << 31)
            | ((exponent + 127) << 23)
            | (mantissa & 0x7fffff)
        );
}
```

Note that this function works for normalized values, denormalized values, and zero, but does not work for NaNs and infinities.

With the utility routines out of the way, take a look at the `fpadd()` function, which adds two floating-point values, producing a 32-bit real result:

```
void fpadd( real left, real right, real *dest )
{
    // The following variables hold the fields associated with the
    // left operand:
    int          Lexponent;
    uint32_t     Lmantissa;
    int          Lsign;

    // The following variables hold the fields associated with the
    // right operand:
    int          Rexponent;
    uint32_t     Rmantissa;
    int          Rsign;

    // The following variables hold the separate fields of the result:
    int          Dexponent;
    uint32_t     Dmantissa;
    int          Dsign;

    // Extract the fields so that they're easy to work with:
    Lexponent = extractExponent( left );
    Lmantissa = extractMantissa( left );
    Lsign     = extractSign( left );

    Rexponent = extractExponent( right );
    Rmantissa = extractMantissa( right );
    Rsign     = extractSign( right );
```

```

// Code to handle special operands (infinity and NaNs):

if( Lexponent == 127 )
{
    if( Lmantissa == 0 )
    {
        // If the left operand is infinity, then the result
        // depends upon the value of the right operand.

        if( Rexponent == 127 )
        {
            // If the exponent is all 1 bits (127 after unbiasing)
            // then the mantissa determines if we have an infinity value
            // (zero mantissa), a QNaN (mantissa = 0x800000), or a SNaN
            // (nonzero mantissa not equal to 0x800000).

            if( Rmantissa == 0 ) // Do we have infinity?
            {
                // infinity + infinity = infinity
                // -infinity - infinity = -infinity
                // -infinity + infinity = NaN
                // infinity - infinity = NaN

                if( Lsign == Rsign )
                {
                    *dest = right;
                }
                else
                {
                    *dest = 0x7fC00000; // +QNaN
                }
            }
            else // Rmantissa is nonzero, so it's a NaN
            {
                *dest = right; // Right is a NaN, propagate it.
            }
        }
    }

    else // Lmantissa is nonzero, Lexponent is all 1s.
    {
        // If the left operand is some NaN, then the result will
        // also be the same NaN.

        *dest = left;
    }

    // We've already calculated the result, so just return.
    return;
}

else if( Rexponent == 127 )
{
    // Two case: right is either a NaN (in which case we need to
    // propagate the NaN regardless of left's value) or it is

```

```

    // +/- infinity. Because left is a "normal" number, we'll also
    // wind up propagating the infinity because any normal number
    // plus infinity is infinity.

    *dest = right; // Right is a NaN, so propagate it.
    return;
}

// Okay, we've got two actual floating-point values. Let's add them
// together. First, we have to "denormalize" one of the operands if
// their exponents aren't the same (when adding or subtracting values,
// the exponents must be the same).
//
// Algorithm: choose the value with the smaller exponent. Shift its
// mantissa to the right the number of bits specified by the difference
// between the two exponents.

Dexponent = Rexponent;
if( Rexponent > Lexponent )
{
    shiftAndRound( &Lmantissa, (Rexponent - Lexponent));
}
else if( Rexponent < Lexponent )
{
    shiftAndRound( &Rmantissa, (Lexponent - Rexponent));
    Dexponent = Lexponent;
}

// Okay, add the mantissas. There is one catch: if the signs are opposite
// then we've actually got to subtract one value from the other (because
// the FP format is one's complement, we'll subtract the larger mantissa
// from the smaller and set the destination sign according to a
// combination of the original sign values and the largest mantissa).

if( Rsign ^ Lsign )
{
    // Signs are different, so we must subtract one value from the other.

    if( Lmantissa > Rmantissa )
    {
        // The left value is greater, so the result inherits the
        // sign of the left operand.

        Dmantissa = Lmantissa - Rmantissa;
        Dsign = Lsign;
    }
    else
    {
        // The right value is greater, so the result inherits the
        // sign of the right operand.

        Dmantissa = Rmantissa - Lmantissa;
        Dsign = Rsign;
    }
}

```

```

    }
    else
    {
        // Signs are the same, so add the values:

        Dsign = Lsign;
        Dmantissa = Lmantissa + Rmantissa;
    }

    // Normalize the result here.
    //
    // Note that during addition/subtraction, overflow of 1 bit is possible.
    // Deal with that possibility here (if overflow occurred, shift the
    // mantissa to the right one position and adjust for this by incrementing
    // the exponent). Note that this code returns infinity if overflow occurs
    // when incrementing the exponent (infinity is a value with an exponent
    // of $FF);

    if( Dmantissa >= 0x1000000 )
    {
        // Never more than 1 extra bit when doing addition/subtraction.
        // Note that by virtue of the floating-point format we're using,
        // the maximum value we can produce via addition or subtraction is
        // a mantissa value of 0x1fffffe. Therefore, when we round this
        // value it will not produce an overflow into the 25th bit.

        shiftAndRound( &Dmantissa, 1 ); // Move result into 24 bits.
        ++Dexponent;                      // Shift operation did a div by 2,
                                         // this counteracts the effect of
                                         // the shift (incrementing exponent
                                         // multiplies the value by 2).
    }
    else
    {
        // If the H0 bit is clear, normalize the result
        // by shifting bits up and simultaneously decrementing
        // the exponent. We will treat 0 as a special case
        // because it's a common enough result.

        if( Dmantissa != 0 )
        {
            // The while loop multiplies the mantissa by 2 (via a shift
            // left) and then divides the whole number by 2 (by
            // decrementing the exponent. This continues until the H0 bit of
            // Dmantissa is set or the exponent becomes -127 (0 in the
            // biased-127 form). If Dexponent drops down to -128, then we've
            // got a denormalized number and we can stop.

            while( (Dmantissa < 0x800000) && (Dexponent > -127 ))
            {
                Dmantissa = Dmantissa << 1;
                --Dexponent;
            }
        }
    }
}

```



```

    }
    else
    {
        // If the mantissa went to 0, clear everything else, too.

        Dsign = 0;
        Dexponent = 0;
    }
}

// Reconstruct the result and store it away:

*dest = packFP( Dsign, Dexponent, Dmantissa );
}

```

To conclude this discussion of the software implementation of the `fpadd()` and `fsub()` functions, here's a C `main()` function demonstrating their use:

```

// A simple main program that does some trivial tests on fpadd and fsub.

int main( int argc, char **argv )
{
    real l, r, d;

    asreal(l) = 1.0;

    asreal(r) = 2.0;

    fpadd( l, r, &d );
    printf( "dest = %x\n", d );
    printf( "dest = %12E\n", asreal( d ));

    l = d;
    asreal(r) = 4.0;
    fsub( l, r, &d );
    printf( "dest2 = %x\n", d );
    printf( "dest2 = %12E\n", asreal( d ));
}

```

Here's the output produced by compiling with Microsoft Visual C++ (and defining `uint32_t` as an unsigned long):

```

l = 3f800000
l = 1.000000E+00
r = 40000000
r = 2.000000E+00
dest = 40400000
dest = 3.000000E+00
dest2 = bf800000
dest2 = -1.000000E+00

```

4.7.3 Floating-Point Multiplication and Division

Most software floating-point libraries are actually written in hand-optimized assembly language, not in a high-level language (HLL). As the previous section shows, it's possible to write floating-point routines in an HLL and, particularly in the case of single-precision floating-point addition and subtraction, you could write the code efficiently. Given the right library routines, you could also write the floating-point multiplication and division routines in an HLL. However, because their implementation is actually easier in assembly language, this section presents an HLA implementation of the single-precision floating-point multiplication and division algorithms.

The HLA code in this section implements two functions, `fpmul()` and `fpdiv()`, that have the following prototypes:

```
procedure fpmul( left:real32; right:real32 ); @returns( "eax" );  
procedure fpdiv( left:real32; right:real32 ); @returns( "eax" );
```

Beyond the fact that this code is written in assembly language rather than C, it differs in two main ways from the code in the previous section. First, it uses the built-in `real32` data type rather than creating a new data type for the real values, because we can easily coerce any 32-bit memory object to `real32` or `dword` in assembly language. Second, these prototypes support only two parameters; there is no destination parameter. These functions simply return the `real32` result in the EAX register.⁷

4.7.3.1 Floating-Point Multiplication

Whenever you multiply two values in scientific notation, you compute the result sign, exponent, and mantissa as follows:

- The result sign is the exclusive-OR of the operand signs. That is, the result is positive if both operand signs were the same, and the result sign is negative if the operand signs were different.
- The result exponent is the sum of the operands' exponents.
- The result mantissa is the integer (fixed-point) product of the two operand mantissas.

There are a few additional rules that affect the floating-point multiplication algorithm that are a direct result of the IEEE floating-point format:

- If either, or both, of the operands are 0, the result is 0 (this is a special case because the representation for 0 is special).

7. Those who know a little 80x86 assembly language may wonder if it's legal to return a floating-point value in an integer register. Indeed, it is! EAX can hold any 32-bit value, not just integers. Presumably, if you're writing a software-based floating-point package, you don't have floating-point hardware available and, therefore, you can't pass floating-point values around in the floating-point registers.

- If either operand is infinity, the result is infinity.
- If either operand is a NaN, the result is that same NaN.

The `fpmul()` procedure begins by checking if either of the operands is 0. If so, the function immediately returns 0.0 to the caller. Next, the `fpmul()` code checks for NaN or infinity values in the left and right operands. If it finds one of these values, it returns that same value to the caller.

If both of the `fpmul()` operands are reasonable floating-point values, then the `fpmul()` code extracts the sign, exponent, and mantissa fields of the packed floating-point value. Actually, *extract* isn't the correct term here; *isolate* is a better description. Here's the code that isolates the sign bits of the two operands and computes the result sign:

```
mov( (type dword left), ebx ); // Result sign is the XOR of the
xor( (type dword right), ebx ); // operand signs.
and( $8000_0000, ebx ); // Keep only the sign bit.
```

This code exclusive-ORs the two operands and then masks out bits 0 through 30, leaving only the result sign value in bit 31 of the EBX register. This procedure doesn't bother moving the sign bit down to bit 0 (as you'd normally do when unpacking data), because it would just have to move this bit back to bit 31 when it repacks the floating-point value later.

To process the exponent, `fpmul()` isolates bits 23 through 30 and operates on the exponent in place. When multiplying two values using scientific notation, you must add the values of the exponents together. However, you must subtract 127 from the exponent's sum, since adding excess-127 exponents ends up adding the bias twice. The following code isolates the exponent bits, adjusts for the extra bias, and adds the exponents together:

```
mov( (type dword left), ecx ); // Exponent goes into bits 23..30
and( $7f80_0000, ecx ); // of ECX; mask these bits.
sub( 126 << 23, ecx ); // Eliminate the bias of 127 and multiply by 2

mov( (type dword right), eax );
and( $7f80_0000, eax );

// For multiplication, we need to add the exponents:

add( eax, ecx ); // Exponent value is now in bits
// 23..30 of ECX.
```

First, notice that this code subtracts 126 rather than 127. The reason is that later we'll need to double the result of the multiplication of the mantissas. Subtracting 126 rather than 127 does this multiplication by 2 implicitly (saving an instruction later on).

If the sum of the exponents with `add(eax, ecx)` in the preceding code is too large to fit into 8 bits, there will be a carry out of bit 30 into bit 31 of ECX, which will set the 80x86 overflow flag. If overflow occurs on a multiplication, our code will return infinity as the result.

If overflow does not occur, then the `fpmul()` procedure needs to set the implied HO bit of the two mantissa values. The following code handles this chore, strips out all the exponent and sign bits from the mantissas, and left-justifies the mantissa bits up against bit position 31 in EAX and EDX.

```
mov( (type dword left), eax );
mov( (type dword right), edx );

// If we don't have a 0 value, then set the implied HO bit of the mantissa:

if( eax <> 0 ) then

    or( $80_0000, eax ); // Set the implied bit to 1.

endif;
shl( 8, eax ); // Moves mantissa to bits 8..31 and removes sign/exp.

// Repeat this for the right operand.

if( edx <> 0 ) then

    or( $80_0000, edx );

endif;
shl( 8, edx );
```

Once the mantissas are shifted to bit 31 in EAX and EDX, we multiply using the 80x86 `mul()` instruction:

```
mul( edx );
```

This instruction computes the 64-bit product of EAX and EDX, leaving the result in EDX:EAX (the HO double word is in EDX, and the LO double word is in EAX). Because the product of any two n -bit integers could require as many as $2 \times n$ bits, the `mul()` instruction computes EDX:EAX = EAX $\times$ EDX. Left-justifying the mantissas in EAX and EDX before doing the multiplication ensures the mantissa of the product winds up in bits 7 through 30 of EDX. We actually need them in bit positions 8 through 31 of EDX—that’s why earlier this code subtracted only 126, rather than 127, when adjusting for the excess-127 value (this multiplies the result by 2, which is equivalent to shifting the bits left one position). As these numbers were normalized prior to the multiplication, bit 30 of EDX will contain a 1 after the multiplication unless the result is 0. The 32-bit IEEE real format does not support denormalized values, so we don’t have to worry about this case when using 32-bit floating-point values.

Because the mantissas are 24 bits each, the product of the mantissas could have as many as 48 significant bits. Our result mantissa can hold only 24 bits, so we need to round the value to produce a 24-bit result (using the

IEEE rounding algorithm — see “Rounding” on page 71). Here’s the code that rounds the value in EDX to 24 significant bits (in positions 8..31):

```
test( $80, edx ); // Clears zero flag if bit 7 of EDX = 1.
if( @nz ) then

    add( $FFFF_FFFF, eax ); // Sets carry if EAX <> 0.
    adc( $7f, dl );          // Sets carry if DL:EAX > $80_0000_0000.
    if( @c ) then

        // If DL:EAX > $80_0000_0000 then round the mantissa
        // up by adding 1 to bit position 8:

        add( 1 << 8, edx );

    else // DL:EAX = $80_0000_0000

        // We need to round to the value that has a 0
        // in bit position 0 of the mantissa (bit #8 of EDX):

        test( 8, edx ); // Clears zero flag if bit #8 contains a 1.
        if( @nz ) then

            add( 1 << 8, edx ); // Adds a 1 starting at bit position 8.

            // If there was an overflow, renormalize:

            if( @c ) then

                rcr( 1, edx ); // Shift overflow (in carry) back into EDX.
                inc( ecx );    // Shift did a divide by 2. Fix that.

            endif;

        endif;

    endif;

endif;
```

The number may need to be renormalized after rounding. If the mantissa contains all 1 bits and needs to be rounded up, this will produce an overflow out of the HO bit of the mantissa. The `rcr()` and `inc()` instructions at the end of this code sequence put the overflow bit back into the mantissa if overflow occurs.

The only thing left to do after this is pack the destination sign, exponent, and mantissa into the 32-bit EAX register. The following code does this:

```
shr( 8, edx );          // Move mantissa into bits 0..23.
and( $7f_ffff, edx );   // Clear the implied bit.
lea( eax, [edx+ecx] );   // Merge mantissa and exponent into EAX.
or( ebx, eax );          // Merge in the sign.
```

The only tricky thing in this code is the use of the `lea()` (load effective address) instruction to compute the sum of `EDX` (the mantissa) and `ECX` (the exponent) and move the result to `EAX` all with a single instruction.

4.7.3.2 Floating-Point Division

Floating-point division is a little bit more involved than multiplication because the IEEE floating-point standard says many things about degenerate conditions that can occur during division. We're not going to discuss all the code that handles those conditions here. Instead, see the discussion of the conditions for `fpmul()` earlier, and check out the complete code listing for `fdiv()` later in this section.

Assuming we have reasonable numbers to divide, the division algorithm first computes the result sign using the same algorithm (and code) as for multiplying. When dividing two values using scientific notation, we have to subtract their exponents. In contrast to the multiplication algorithm, here it's more convenient to truly unpack the exponents for the two division operands and convert them from excess-127 to two's complement form. Here's the code that does this:

```
mov( (type dword left), ecx ); // Exponent comes from bits 23..30.
shr( 23, ecx );
and( $ff, ecx );                // Mask out the sign bit (in bit 8).

mov( (type dword right), eax );
shr( 23, eax );
and( $ff, eax );

// Eliminate the bias from the exponents:

sub( 127, ecx );
sub( 127, eax );

// For division, we need to subtract the exponents:

sub( eax, ecx );                // Leaves result exponent in ECX.
```

The 80x86 `div()` instruction absolutely, positively requires the quotient to fit into 32 bits. If this condition is not true, the CPU may abort the operation with a divide exception. As long as the HO bit of the divisor contains a 1 and the HO 2 bits of the dividend contain %01, we won't get a division error. Here's the code that prepares the operands prior to the division operation:

```
mov( (type dword left), edx );
if( edx <> 0 ) then

    or( $80_0000, edx ); // Set the implied bit to 1 in the left operand.
    shl( 8, edx );

endif;
mov( (type dword right), edi );
```

```

if( edi <> 0 ) then

    or( $80_0000, edi );      // Set the implied bit to 1 in the right operand.
    shl( 8, edi );

else

    // Division by zero error, here.

endif;

```

The next step is to actually do the division. As noted earlier, in order to prevent a division error, we have to shift the dividend 1 bit to the right (to set the HO 2 bits to %01), as follows:

```

xor( eax, eax );    // EAX := 0;
shr( 1, edx );      // Shift EDX:EAX to the right 1 bit to
rcr( 1, eax );      // prevent a division error.
div( edi );         // Compute EAX = EDX:EAX / EDI.

```

Once the `div()` instruction executes, the quotient is sitting in the HO 24 bits of EAX, and the remainder is in AL:EDX. We now need to normalize and round the result. Rounding is a little easier because AL:EDX contains the remainder after the division; if we need to round down, it will contain a value less than \$80:0000_0000 (that is, the 80x86 AL register contains \$80 and EDX contains 0); if we need to round up, it will contain a value greater than \$80:0000_; and if we need to round to the nearest value, it will contain exactly \$80:0000_0000.

Here's the code that does this:

```

test( $80, al );    // See if the bit just below the LO bit of the
if( @nz ) then      // mantissa contains a 0 or 1.

    // Okay, the bit just below the LO bit of our mantissa contains a 1.
    // If all other bits below the mantissa and this bit contain 0s,
    // we have to round to the nearest mantissa value whose LO bit is 0.

    test( $7f, al );      // Clears zero flag if bits 0..6 <> 0.
    if( @nz || edx <> 0 ) then // If bits 0..6 in AL are 0 and EDX
                                // is 0.

        // We need to round up:

        add( $100, eax ); // Mantissa starts in bit #8 );
        if( @c ) then      // Carry set if mantissa overflows.

            // If there was an overflow, renormalize.

            rcr( 1, eax );
            inc( ecx );

endif;

```

```

else

    // The bits below the mantissa are exactly 1/2 the value
    // of the LO mantissa bit. So we need to round to the value
    // that has a LO mantissa bit of 0:

    test( $100, eax );
    if( @nz ) then

        add( $100, eax );
        if( @c ) then

            // If there was an overflow, renormalize.

            rcr( 1, eax ); // Put overflow bit back into EAX.
            inc( ecx );    // Adjust exponent accordingly.

        endif;

    endif;

endif;

endif;
endif;

```

The last step in `fpdiv` is to add the bias back into the exponent (and verify that overflow doesn't occur) and then pack the quotient's sign, exponent, and mantissa fields into the 32-bit floating-point format. Here's the code that does this:

```

if( (type int32 ecx) > 127 ) then

    mov($ff-127, ecx ); // Set exponent value for infinity
    xor( eax, eax );   // because we just had overflow.

elseif( (type int32 ecx) < -128 ) then

    mov( -127, ecx ); // Return 0 for underflow (note that
    xor( eax, eax ); // next we add 127 to ECX).

endif;
add( 127, ecx );      // Add the bias back in.
shl( 23, ecx );      // Move the exponent to bits 23..30.

// Okay, assemble the final real32 value:

shr( 8, eax );        // Move mantissa into bits 0..23.
and( $7f_ffff, eax ); // Clear the implied bit.
or( ecx, eax );       // Merge mantissa and exponent into EAX.
or( ebx, eax );       // Merge in the sign.

```

Whew! This has been a lot of code. However, going through all of it just to see how floating-point operations work has hopefully given you an appreciation of exactly what an FPU does for you.

4.8 For More Information

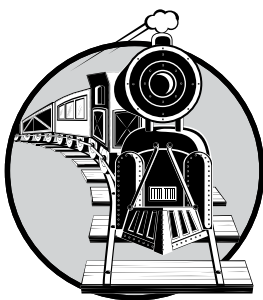
Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

———. “Webster: The Place on the Internet to Learn Assembly.” <http://plantation-productions.com/Webster/index.html>.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Boston: Addison-Wesley, 1998.

5

CHARACTER REPRESENTATION



Although computers are famous for their “number-crunching” capabilities, the truth is that most computer systems process character data far more often than numbers. The term *character* refers to a human- or machine-readable symbol that is typically a non-numeric entity.

In general, a character is any symbol that you can type on a keyboard or show on a video display. In addition to alphabetic characters, character data includes punctuation marks, numeric digits, spaces, tabs, carriage returns (the ENTER key), other control characters, and other special symbols.

This chapter looks at how to represent characters, strings, and character sets within a computer system. It also discusses various operations on these data types.

5.1 Character Data

Most computer systems use a 1-byte or multibyte binary sequence to encode the various characters. Windows, macOS, and Linux fall into this category, using the ASCII or Unicode character sets, whose members can all be represented with 1- or multibyte binary sequences. The EBCDIC character set, in use on IBM mainframes and minicomputers, is another example of a single-byte character code.

This chapter will discuss all three of these character sets and their internal representations, as well as how to create your own character sets.

5.1.1 The ASCII Character Set

The ASCII (American Standard Code for Information Interchange) character set maps 128 characters to the unsigned integer values 0 through 127 (\$0 through \$7F). Although the exact mapping of characters to numeric values is arbitrary and unimportant, a standardized mapping allows you to communicate between programs and peripheral devices. The standard ASCII codes are useful because nearly everyone uses them. If you use the ASCII code 65 to represent the character *A*, for example, you can be confident that some peripheral device (such as a printer) will correctly interpret this value as an *A*.

Because the ASCII character set provides only 128 different characters, you might be wondering: “What do we do with the additional 128 values (\$80..\$FF) that we can represent with a byte?” One option is to ignore those extra values, and that’s the primary approach of this book. Another possibility is to extend the ASCII character set by an additional 128 characters. Of course, unless you can get everyone to agree upon a particular extension of the character set¹ (a difficult task indeed), the whole purpose of having a standardized character set will be defeated.

Despite some major shortcomings, such as the inability to represent all characters and alphabets in use today, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data, and most programs can produce it. Because you’ll probably be dealing with ASCII characters in your programs, it would be wise to study the layout of the character set and memorize a few key ASCII codes (such as those for *0*, *A*, and *a*).

NOTE

Table A-1 in Appendix A lists all the characters in the standard ASCII character set.

The ASCII character set is divided into four groups of 32 characters. The first 32 characters, ASCII codes \$0 through \$1F (0 through 31), form a special set of nonprinting characters called the *control characters*. As their name implies, these characters perform various printer and display control operations rather than displaying symbols. Examples of control characters

1. Back before Windows became popular, IBM supported an extended 256-element character set on its text displays. Though this character set is “standard” even on modern PCs, few applications or peripheral devices continue to use it.

include the carriage return, which positions the cursor at the beginning of the current line of characters;² line feed, which moves the cursor down one line on the output device; and backspace, which moves the cursor back one position to the left. Unfortunately, because there's very little standardization among output devices, different control characters perform different operations on different output devices. To find out exactly how a particular control character affects a certain device, consult the device's manual.

The second group of 32 ASCII character codes comprises various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code \$20) and the numeric digits (ASCII codes \$30..\$39).

The third group of 32 ASCII characters contains the uppercase alphabetic characters. The ASCII codes for the characters A through Z lie in the range \$41 through \$5A. Because there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth and final group of 32 ASCII character codes represents the lowercase alphabetic symbols, five additional special symbols, and another control character (delete). The lowercase character symbols use the ASCII codes \$61 through \$7A. If you convert the codes for the upper- and lowercase characters to binary, you'll notice that the uppercase symbols differ from their lowercase equivalents in exactly one bit position. For example, consider the character codes for *E* and *e* in Figure 5-1.



Figure 5-1: ASCII codes for *E* and *e*

These two codes differ only in bit 5. Uppercase alphabetic characters always contain a 0 in bit 5; lowercase alphabetic characters always contain a 1 in bit 5. To quickly convert an alphabetic character between upper- and lowercase, simply invert bit 5. To force an uppercase character to lowercase, set bit 5 to 1. Likewise, you can force a lowercase character to uppercase by setting bit 5 to 0.

Bits 5 and 6 determine the character's group (see Table 5-1). Therefore, you can convert any upper- or lowercase (or special) character to its corresponding control character by setting bits 5 and 6 to 0 (for example, *A* becomes CTRL-A when you set bits 5 and 6 to 0; that is, 0x41 becomes 0x01).

2. Historically, *carriage return* refers to the paper carriage used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left-hand side of the paper.

Table 5-1: ASCII Character Groups Determined by Bits 5 and 6

Bit 6	Bit 5	Group
0	0	Control characters
0	1	Digits and punctuation
1	0	Uppercase and special
1	1	Lowercase and special

Bits 5 and 6 aren't the only bits that encode useful information. Consider, for a moment, the ASCII codes of the numeric digit characters in Table 5-2. The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation reveals something very important—the LO nibble is the binary equivalent of the represented number. By stripping away (setting to 0) the HO nibble of the ASCII code, you obtain the binary representation of that digit. Conversely, you can convert a binary value in the range 0 through 9 to its ASCII character representation by simply setting the HO nibble to %0011, or the decimal value 3. You can use the logical AND operation to force the HO bits to 0; likewise, you can use the logical OR operation to force the HO bits to %0011. For more information on string-to-numeric conversions, see Chapter 2.

Table 5-2: ASCII Codes for the Numeric Digits

Character	Decimal	Hexadecimal
0	48	\$30
1	49	\$31
2	50	\$32
3	51	\$33
4	52	\$34
5	53	\$35
6	54	\$36
7	55	\$37
8	56	\$38
9	57	\$39

Despite the fact that it is a “standard,” simply encoding your data using ASCII characters does not guarantee compatibility across systems. An *A* on one machine is most likely an *A* on another system; but, of the 32 control codes in the first group of ASCII codes, plus the delete code in the last group, only 4 control codes are commonly supported by most devices and applications—backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these “supported” control codes in different ways. End-of-line is a particularly troublesome example. Windows, MS-DOS, CP/M, and other systems mark end-of-line by the two-character sequence CR/LF. The original Apple Macintosh OS and many other systems

mark end-of-line by a single CR character. Linux, BeOS, macOS, and other Unix systems mark end-of-line with a single LF character.

Exchanging simple text files between different systems can be an exercise in frustration. Even if you use standard ASCII characters in all your files, you still need to convert the data when exchanging files between systems. Fortunately, many text editors automatically handle files with different line endings (many available freeware utilities will also do this conversion for you). If you have to do this in your own software, simply copy all characters except the end-of-line sequence from one file to another, and then emit the new end-of-line sequence whenever you encounter an old end-of-line sequence in the input file.

5.1.2 The EBCDIC Character Set

Although the ASCII character set is, unquestionably, the most popular character representation, it's certainly not the only one available. For example, IBM uses the *EBCDIC* code on many of its mainframe and mini-computer lines. However, you'll rarely encounter it on personal computer systems, so we'll consider it only briefly in this book.

EBCDIC (pronounced "Eb-suh-dic") stands for *Extended Binary Coded Decimal Interchange Code*. If you're wondering whether there was an unextended version of this character code, the answer is yes. Earlier IBM systems and keypunch machines used *BCDIC* (*Binary Coded Decimal Interchange Code*), a character set based on punched cards and decimal representation (for IBM's older decimal machines).

BCDIC existed long before modern digital computers; it was born on old-fashioned IBM keypunches and tabulator machines. EBCDIC extended that encoding to provide a character set for IBM's computers. However, EBCDIC inherited several traits from BCDIC that seem strange in the context of modern computers. For example, the encodings of the alphabetic characters are not contiguous. Originally, the alphabetic characters probably did have a sequential encoding; however, when IBM expanded the character set, it used some binary combinations that aren't present in the BCD format (like %1010...%1111). These binary values appear between two otherwise sequential BCD values, which explains why certain character sequences (such as the alphabetic characters) aren't contiguous in the EBCDIC encoding.

EBCDIC is not a single character set; rather, it is a family of character sets. While the EBCDIC character sets have a common core (for example, the encodings for the alphabetic characters are usually the same), different versions, known as *code pages*, have different encodings for punctuation and special characters. Because of the limited number of encodings available in a single byte, different code pages reuse some of the character encodings for their own special set of characters. So, if you're given a file that contains EBCDIC characters and someone asks you to translate it to ASCII, you'll quickly discover that it's not a trivial task.

Because of the weirdness of the EBCDIC character set, many common algorithms that work well on ASCII characters simply don't work with

EBCDIC. However, keep in mind that EBCDIC functional equivalents exist for most ASCII characters. Check out the IBM literature for more details.

5.1.3 Double-Byte Character Sets

Because a byte can represent a maximum of 256 characters, some computer systems use *double-byte character sets (DBCSs)* to represent more than 256 characters. DBCSs do not encode every character using 16 bits; instead, they use a single byte for most character encodings and use double-byte codes only for certain characters.

A typical double-byte character set uses the standard ASCII character set along with several additional characters in the range \$80 through \$FF. Certain values in this range are used as extension codes that tell the software that a second byte immediately follows. Each extension byte allows the DBCS to support another 256 different character codes. With three extension values, for example, the DBCS can support up to 1,021 different characters: 256 characters for each of the extension bytes, and 253 ($256 - 3$) characters for the standard single-byte set (we subtract 3 because the three extension byte values each consume one of the 256 combinations, and they don't count as characters).

Back in the days when terminals and computers used memory-mapped character displays, double-byte character sets weren't very practical. Hardware character generators really want each character to be the same size, and they want to process a limited number of characters. However, as bitmapped displays with software character generators became prevalent (such as Windows, Macintosh, Unix/XWindows machines, tablets, and smartphones), it became possible to process DBCSs.

Although DBCSs can compactly represent a large number of characters, more computing resources are required to process text in a DBCS format. For example, determining the length of a zero-terminated string containing DBCS characters (typical in the C/C++ languages) can be considerable work. Some characters in the string consume 2 bytes, while most others consume only 1 byte, so a string length function has to scan the string byte-by-byte to locate any extension values indicating that a single character consumes 2 bytes. This process more than doubles the time a high-performance string length function takes to execute.

Worse still, many common algorithms used to manipulate string data fail when applied to DBCSs. For example, a common C/C++ trick to step through characters in a string is to either increment or decrement a pointer to the string using expressions like `++ptrChar` or `--ptrChar`. This won't work with DBCSs. While someone using a DBCS probably has a set of standard C library routines that work on DBCSs, it's also quite likely that other character functions they or others have written don't work properly with the extended characters.

The other big problem with DBCSs is the lack of consistent standard. Different DBCSs use the same exact encoding for different characters. For these reasons, if you need a standardized character set that supports more than 256 characters, you're far better off using the Unicode character set.

5.1.4 The Unicode Character Set

A few decades back, engineers at Aldus, NeXT, Sun, Apple Computer, IBM, Microsoft, the Research Library Group, and Xerox realized that their new computer systems with bitmaps and user-selectable fonts could display far more than 256 different characters at one time. At the time, DBCSs were the most common solution, but—as just noted—they had a couple of compatibility problems. So, the engineers sought a different route.

The solution they came up with was the Unicode character set. The engineers who originally developed Unicode chose a 2-byte character size. Like DBCSs, this approach still required special library code (existing single-byte string functions would not always work with double-byte characters), but other than changing the size of a character, most existing string algorithms would still work with 2-byte characters. The Unicode definition included all of the (known/living) character sets at the time, giving each character a unique encoding, to avoid the consistency problems that plagued differing DBCSs.

The original Unicode standard used a 16-bit word to represent each character. Therefore, Unicode supported up to 65,536 different character codes—a huge advance over the 256 possible codes that are representable with an 8-bit byte. Furthermore, Unicode is upward compatible from ASCII. If the HO 9 bits³ of a Unicode character's binary representation contain 0, then the LO 7 bits use the standard ASCII code. If the HO 9 bits contain some nonzero value, then the 16 bits form an extended character code (extended from ASCII, that is). If you're wondering why so many different character codes are necessary, note that, at the time, certain Asian character sets contained 4,096 characters. The Unicode character set even provided a set of codes you could use to create an application-defined character set. Approximately half of the 65,536 possible character codes have been defined, and the remaining character encodings are reserved for future expansion.

Today, Unicode is a universal character set, long replacing ASCII and older DBCSs. All modern operating systems (including macOS, Windows, Linux, iOS, Android, and Unix), web browsers, and most modern applications provide Unicode support. Unicode Consortium, a nonprofit corporation, maintains the Unicode standard. By maintaining the standard, Unicode, Inc. (<https://home.unicode.org/>), helps guarantee that a character you write on one system will display as you expect on a different system or application.

5.1.5 Unicode Code Points

Alas, as well thought-out as the original Unicode standard was, it couldn't have anticipated the explosion in characters that would occur. Emojis, astrological symbols, arrows, pointers, and a wide variety of symbols introduced for the internet, mobile devices, and web browsers have greatly

3. ASCII is a 7-bit code. If the HO 9 bits of a 16-bit Unicode value are all 0, the remaining 7 bits are an ASCII encoding for a character.

expanded the Unicode symbol repertoire (along with a desire to support historic, obsolete, and rare scripts). In 1996, systems engineers discovered that 65,536 symbols were insufficient. Rather than require 3 or 4 bytes for each Unicode character, those in charge of the Unicode definition gave up on trying to create a fixed-size representation of characters and allowed for opaque (and multiple) encodings of Unicode characters. Today, Unicode defines 1,112,064 code points, far exceeding the 2-byte capacity originally set aside for Unicode characters.

A Unicode *code point* is simply an integer value that Unicode associates with a particular character symbol; you can think of it as the Unicode equivalent of the ASCII code for a character. The convention for Unicode code points is to specify the value in hexadecimal with a U+ prefix; for example, U+0041 is the Unicode code point for the letter A.

NOTE

See https://en.wikipedia.org/wiki/Unicode#General_Category_property_for_more_details_on_code_points.

5.1.6 Unicode Code Planes

Because of its history, blocks of 65,536 characters are special in Unicode—they are known as a *multilingual plane*. The first multilingual plane, U+000000 to U+00FFFF, roughly corresponds to the original 16-bit Unicode definition; the Unicode standard calls this the *Basic Multilingual Plane (BMP)*. Planes 1 (U+010000 to U+01FFFF), 2 (U+020000 to U+02FFFF), and 14 (U+0E0000 to U+0EFFFF) are supplementary planes. Unicode reserves planes 3 through 13 for future expansion and planes 15 and 16 for user-defined character sets.

The Unicode standard defines code points in the range U+000000 to U+10FFFF. Note that 0x10ffff is 1,114,111, which is where most of the 1,112,064 characters in the Unicode character set come from; the remaining 2,048 code points are reserved for use as *surrogates*, which are Unicode extensions. *Unicode scalar*, another term you might hear, is a value from the set of all Unicode code points *except* the 2,048 surrogate code points. The HO two hexadecimal digits of the six-digit code point value specify the multilingual plane. Why 17 planes? The reason, as you'll see in a moment, is that Unicode uses special multiword entries to encode code points beyond U+FFFF. Each of the two possible extensions encodes 10 bits, for a total of 20 bits; 20 bits gives you 16 multilingual planes, which, plus the original BMP, produces 17 multilingual planes. This is also why code points fall in the range U+000000 to U+10FFFF: it takes 21 bits to encode the 16 multilingual planes plus the BMP.

5.1.7 Surrogate Code Points

As noted earlier, Unicode began life as a 16-bit (2-byte) character set encoding. When it became apparent that 16 bits were insufficient to handle all the possible characters that existed at the time, an expansion was necessary. As of Unicode v2.0, the Unicode, Inc., organization extended the definition of Unicode to include multiword characters. Now Unicode uses surrogate

code points (U+D800 through U+DFFF) to encode values larger than U+FFFF. Figure 5-2 shows the encoding.

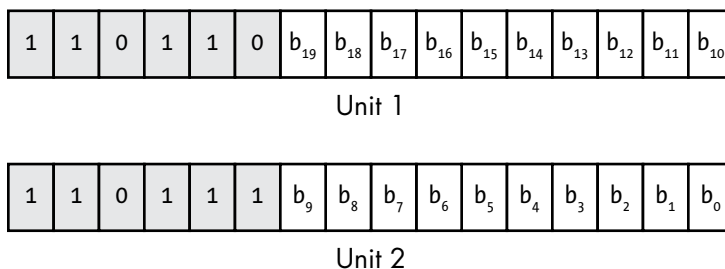


Figure 5-2: Surrogate code point encoding for Unicode planes 1–16

Note that the two words (unit 1/high surrogate and unit 2/low surrogate) always appear together. The unit 1 value (with HO bits %110110) specifies the upper 10 bits ($b_{10}..b_{19}$) of the Unicode scalar, and the unit 2 value (with HO bits %110111) specifies the lower 10 bits ($b_0..b_9$) of the Unicode scalar. Therefore, the value of bits b_{16} through b_{19} plus 1 specifies Unicode plane 1 through 16. Bits b_0 through b_{15} specify the Unicode scalar value within the plane.

Note that surrogate codes appear only in the BMP. None of the other multilingual planes contain surrogate codes. Bits b_0 through b_{19} , extracted from the unit 1 and 2 values, always specify a Unicode scalar value (even if the values fall in the range U+D800 through U+DFFF).

5.1.8 Glyphs, Characters, and Grapheme Clusters

Each Unicode code point has a unique name. For example, U+0045 has the name “LATIN CAPITAL LETTER A.” Note that the symbol *A* is *not* the name of the character. *A* is a *glyph*—a series of strokes (one horizontal and two slanted strokes) that a device draws in order to represent the character.

There are many different glyphs for the single Unicode character “LATIN CAPITAL LETTER A.” For example, a Times Roman letter *A* and a Times Roman Italic letter *A* have different glyphs, but Unicode doesn’t differentiate between them (or between *A* characters in any two different fonts). The character “LATIN CAPITAL LETTER A” remains U+0045 regardless of the font or style you use to draw it.

As an interesting side note, if you have access to the Swift programming language, you can print the name of any Unicode character using the following code:

```
import Foundation
let charToPrintName :String = "A"           // Print name of this character

let unicodeName =
    String(charToPrintName).applyingTransform(
        StringTransform(rawValue: "Any-Name"),
        reverse: false
    )! // Forced unwrapping is legit here because it always succeeds.
```

```
print( unicodeName )
```

Output from program:

```
\N{LATIN CAPITAL LETTER A}
```

So, what exactly is a character in Unicode? Unicode scalars are Unicode characters, but there's a difference between what you'd normally call a character and the definition of a scalar. For example, is *é* one character or two? Consider the following Swift code:

```
import Foundation
let eAccent :String = "e\u{301}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

"`\u{301}`" is the Swift syntax for specifying a Unicode scalar value within a string; in this particular case 301 is the hexadecimal code for the *combining acute accent* character.

The first print statement:

```
print( eAccent )
```

prints the character (producing *é* on the output, as we expect).

The second print statement prints the number of characters Swift determines are present in the string:

```
print( "eAccent.count=\(eAccent.count)" )
```

This prints 1 to the standard output.

The third print statement prints the number of elements (UTF-16 elements⁴) in the string:

```
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

This prints 2 on the standard output, because the string holds two words of UTF-16 data.

So, again, is this one character or two? Internally (assuming UTF-16 encoding), the computer sets aside 4 bytes of memory for this single character (two 16-bit Unicode scalar values).⁵ On the screen, however, the output takes only one character position and looks like a single character to the user. When this character appears within a text editor and the cursor is immediately to the right of the character, the user expects that pressing the backspace key will delete it. From the user's perspective, then, this is a single character (as Swift reports when you print the count attribute of the string).

4. See "Unicode Encodings" on page 107 for a discussion of UTF-16 encoding.

5. Swift 5 switches the preferred encoding of strings from UTF-16 to UTF-8; see <https://swift.org/blog/utf8-string/>.

In Unicode, however, a character is largely equivalent to a code point. This is not what people normally think of as a character. In Unicode terminology, a *grapheme cluster* is what people commonly call a character—it's a sequence of one or more Unicode code points that combine to form a single language element (that is, a single character). So, when we talk about characters with respect to symbols that an application displays to an end user, we're really talking about grapheme clusters.

Grapheme clusters can make life miserable for software developers. Consider the following Swift code (a modification of the earlier example):

```
import Foundation
let eAccent :String = "e\u{301}\u{301}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

This code produces the same é and 1 outputs from the first two print statements. The following produces é:

```
print( eAccent )
```

and this print statement produces 1.

```
print( "eAccent.count=\(eAccent.count)" )
```

However, the third print statement:

```
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

displays 3 rather than 2 (as in the original example).

There are definitely three Unicode scalar values in this string (U+0065, U+0301, and U+0301). When printing, the operating system combines the e and the two acute accent combining characters to form the single character é and then outputs the character to the standard output device. Swift is smart enough to know that this combination creates a single output symbol on the display, so printing the result of the `count` attribute continues to output 1. However, there are (undeniably) three Unicode code points in this string, so printing `utf16.count` produces 3 on output.

5.1.9 Unicode Normals and Canonical Equivalence

The Unicode character *é* actually existed on personal computers long before Unicode came along. It's part of the original IBM PC character set and also part of the Latin-1 character set (used, for example, on old DEC terminals). As it turns out, Unicode uses the Latin-1 character set for the code points in the range U+00A0 to U+00FF, and U+00E9 just happens to correspond to the *é* character. Therefore, we can modify the earlier program as follows:

```
import Foundation
let eAccent :String = "\u{E9}"
```

```
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

The outputs from this program are:

```
é
1
1
```

Ouch! Three different strings all producing é but containing a different number of code points. Imagine how this complicates programming strings containing Unicode characters. For example, if you have the following three strings (Swift syntax) and you try to compare them, what will the result be?

```
let eAccent1 :String = "\u{E9}"
let eAccent2 :String = "e\u{301}"
let eAccent3 :String = "e\u{301}\u{301}"
```

To the user, all three strings look the same on the screen. However, they clearly contain different values. If you compare them to see if they are equal, will the result be true or false?

Ultimately, that depends upon whose string libraries you're using. Most current string libraries would return `false` if you compared these strings for equality. Interestingly enough, Swift will claim that `eAccent1` is equal to `eAccent2`, but it isn't smart enough to report that `eAccent1` is equal to `eAccent3` or that `eAccent2` is equal to `eAccent3`—despite the fact that it displays the same symbol for all three strings. Many languages' string libraries simply report that all three strings are unequal.

The three Unicode/Swift strings `"\{E9}"`, `"e\{301}"`, and `"e\{301}\{301}"` all produce the same output on the display; therefore, they are canonically equivalent according to the Unicode standard. Some string libraries won't report any of these strings as being equivalent, however. Others, like the one for Swift, will handle small canonical equivalences (such as `"\{E9}" == "e\{301}"`) but not arbitrary sequences that should be equivalent.⁶

Unicode defines *normal forms* for Unicode strings. One aspect of normal form is to replace canonically equivalent sequences with an equivalent sequence—for example, replace `"e\u{309}"` by `"\u{E9}"` or replace `"\u{E9}"` by `"e\u{309}"` (usually, the shorter form is preferable). Some Unicode sequences allow multiple combining characters. Often, the order of the combining characters is irrelevant to producing the desired grapheme cluster. However, it's easier to compare two such strings if the combining characters are in a specified order. Normalizing Unicode strings may also produce results whose combining characters always appear in a fixed order (thereby improving efficiency of string comparisons).

6. This is probably a good balance of correctness versus efficiency; it can be computationally expensive to handle all the weird cases that won't normally happen, such as `"e\{301}\{301}"`.

5.1.10 Unicode Encodings

As of Unicode v2.0, the standard supports a 21-bit character space capable of handling over a million characters (though most of the code points remain reserved for future use). Rather than use a fixed-size 3-byte (or worse, 4-byte) encoding to allow the larger character set, Unicode, Inc., allows different encodings—UTF-32, UTF-16, and UTF-8—each with its own advantages and disadvantages.⁷

UTF-32 uses 32-bit integers to hold Unicode scalars. The advantage to this scheme is that a 32-bit integer can represent every Unicode scalar value (which requires only 21 bits). Programs that require random access to characters in strings—without having to search for surrogate pairs—and other constant-time operations are (mostly) possible with UTF-32. The obvious drawback to UTF-32 is that each Unicode scalar value requires 4 bytes of storage—twice that of the original Unicode definition and four times that of ASCII characters. It may seem that using two or four times as much storage (over ASCII and the original Unicode) is a small price to pay. After all, modern machines have several orders of magnitude more storage than they did when Unicode first appeared. However, that extra storage has a huge impact on performance, because those additional bytes quickly consume cache storage. Furthermore, modern string processing libraries often operate on character strings 8 bytes at a time (on 64-bit machines). With ASCII characters, that means a given string function can process up to eight characters concurrently; with UTF-32, that same string function can operate on only two characters concurrently. As a result, the UTF-32 version will run four times slower than the ASCII version. Ultimately, even Unicode scalar values are insufficient to represent all Unicode characters (that is, many Unicode characters require a sequence of Unicode scalars), so using UTF-32 doesn't solve the problem.

The second encoding format the Unicode supports is UTF-16. As the name suggests, UTF-16 uses 16-bit (unsigned) integers to represent Unicode values. To handle scalar values greater than 0xFFFF, UTF-16 uses the surrogate pair scheme to represent values in the range 0x010000 to 0x10FFFF (see “Surrogate Code Points” on page 102). Because the vast majority of useful characters fit into 16 bits, most UTF-16 characters require only 2 bytes. For those rare cases where surrogates are necessary, UTF-16 requires 2 words (32 bits) to represent the character.

The last encoding, and unquestionably the most popular, is UTF-8. The UTF-8 encoding is forward compatible from the ASCII character set. In particular, all ASCII characters have a single-byte representation (their original ASCII code, where the HO bit of the byte containing the character contains a 0 bit). If the UTF-8 HO bit is 1, then UTF-8 requires between 1 and 3 additional bytes to represent the Unicode code point. Table 5-3 provides the UTF-8 encoding schema.

7. UTF stands for “Unicode Transformational Format.”

Table 5-3: UTF Encoding

Bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+00	U+7F	0xxxxxxx			
2	11	U+80	U+7FF	110xxxxx	10xxxxxx		
3	16	U+800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The “xxx . . .” bits are the Unicode code point bits. For multibyte sequences, byte 1 contains the HO bits, byte 2 contains the next HO bits (LO bits compared to byte 1), and so on. For example, the 2-byte sequence (%11011111, %10000001) corresponds to the Unicode scalar %0000_0111_1100_0001 (U+07C1).

UTF-8 encoding is probably the most common encoding in use. Most web pages use it. Most C standard library string functions will operate on UTF-8 text without modification (although some C standard library functions can produce malformed UTF-8 strings if the programmer isn’t careful with them).

Different languages and operating systems use different encodings as their default. For example, macOS and Windows tend to use UTF-16 encoding, whereas most Unix systems use UTF-8. Some variants of Python use UTF-32 as their native character format. By and large, though, most programming languages use UTF-8 because they can continue to use older ASCII-based character processing libraries to process UTF-8 characters. Apple’s Swift is one of the first programming languages that attempts to do Unicode right (though there is a huge performance hit for doing so).

5.1.11 Unicode Combining Characters

Although UTF-8 and UTF-16 encodings are much more compact than UTF-32, the CPU overhead and algorithmic complexities of dealing with multi-byte (or multiword) characters sets complicates their use, introducing bugs and performance issues. Despite the issues of wasting memory (especially in the cache), why not simply define characters as 32-bit entities and be done with it? This seems like it would simplify string processing algorithms, improving performance and reducing the likelihood of defects in the code.

The problem with this theory is that you cannot represent all possible grapheme clusters with only 21 bits (or even 32 bits) of storage. Many grapheme clusters consist of several concatenated Unicode code points. Here’s an example from Chris Eidhof and Ole Begemann’s *Advanced Swift* (CreateSpace, 2017):

```
let chars: [Character] = [
    "\u{1ECD}\u{300}",
    "\u{F2}\u{323}",
    "\u{6F}\u{323}\u{300}",
    "\u{6F}\u{300}\u{323}"
]
```

Each of these Unicode grapheme clusters produces an identical character: an ó with a dot underneath the character (this is a character from the Yoruba character set). The character sequence (U+1ECD, U+300) is an o with a dot under it followed by a combining acute. The character sequence (U+F2, U+323) is an ó followed by a combining dot. The character sequence (U+6F, U+323, U+300) is an o followed by a combining dot, followed by a combining acute. Finally, the character sequence (U+6F, U+300, U+323) is an o followed by a combining acute, followed by a combining dot. All four strings produce the same output. Indeed, the Swift string comparisons treat all four strings as equal:

```
print( "\u{1ECD} + \u{300} = \u{1ECD}\u{300}" )
print( "\u{F2} + \u{323} = \u{F2}\u{323}" )
print( "\u{6F} + \u{323} + \u{300} = \u{6F}\u{323}\u{300}" )
print( "\u{6F} + \u{300} + \u{323} = \u{6F}\u{300}\u{323}" )
print( chars[0] == chars[1] ) // Outputs true
print( chars[0] == chars[2] ) // Outputs true
print( chars[0] == chars[3] ) // Outputs true
print( chars[1] == chars[2] ) // Outputs true
print( chars[1] == chars[3] ) // Outputs true
print( chars[2] == chars[3] ) // Outputs true
```

Note that there is not a single Unicode scalar value that will produce this character. You must combine at least two Unicode scalars (or as many as three) to produce this grapheme cluster on the output device. Even if you used UTF-32 encoding, it would still require two (32-bit) scalars to produce this particular output.

Emojis present another challenge that can't be solved using UTF-32. Consider the Unicode scalar U+1F471. This prints an emoji of a person with blond hair. If we add a skin color modifier to this, we obtain (U+1F471, U+1F3FF), which produces a person with a dark skin tone (and blond hair). In both cases we have a single character displaying on the screen. The first example uses a single Unicode scalar value, but the second example requires two. There is no way to encode this with a single UTF-32 value.

The bottom line is that certain Unicode grapheme clusters will require multiple scalars, no matter how many bits we assign to the scalar (it's possible to combine 30 or 40 scalars into a single grapheme cluster, for example). That means we're stuck dealing with multiword sequences to represent a single "character" regardless of how hard we try to avoid it. This is why UTF-32 has never really taken off. It doesn't solve the problem of random access into a string of Unicode characters. If you've got to deal with normalizing and combining Unicode scalars, it's more efficient to use UTF-8 or UTF-16 encodings.

Again, most languages and operating systems today support Unicode in one form or another (typically using UTF-8 or UTF-16 encoding). Despite the obvious problems with dealing with multibyte character sets, modern programs need to deal with Unicode strings rather than simple ASCII strings. Swift, which is almost "pure Unicode," doesn't even offer much in the way of standard ASCII character support.

5.2 Character Strings

After integers, character strings are probably the most common type in use in modern programs. In general, a *character string* is a sequence of characters with two main attributes: a *length* and the *character data*.

Character strings may also possess other attributes, such as the *maximum length* allowable for that particular variable or a *reference count* specifying how many different string variables refer to the same character string. We'll look at these attributes and how programs can use them in this section, which describes various string formats and some of the possible string operations.

5.2.1 Character String Formats

Different languages use different data structures to represent strings. Some string formats use less memory, others allow faster processing, some are more convenient to use, and still others provide additional functionality for the programmer and operating system. To help you better understand the reasoning behind the design of character strings, let's look at some common string representations popularized by various high-level languages.

5.2.1.1 Zero-Terminated Strings

Without question, *zero-terminated strings* are the most common string representation in use today, because this is the native string format for C, C++, and several other languages. In addition, you'll find zero-terminated strings in programs written in languages that don't have a specific native string format, such as assembly language.

A zero-terminated ASCII string is a sequence containing zero or more 8-bit character codes ending with a byte containing 0 (or, in the case of UTF-16, a sequence containing zero or more 16-bit character codes and ending with a 16-bit word containing 0). For example, in C/C++, the ASCII string "abc" requires 4 bytes: 1 byte for each of the three characters a, b, and c, and a 0 byte.

Zero-terminated strings have a few advantages over other string formats:

- Zero-terminated strings can represent strings of any practical length with only one byte of overhead (2 bytes in UTF-16, 4 in UTF-32).
- Given the popularity of the C/C++ programming languages, high-performance string processing libraries are available that work well with zero-terminated strings.
- Zero-terminated strings are easy to implement. As far as the C and C++ languages are concerned, strings are just arrays of characters. That's probably why C's designers chose this format in the first place—so they wouldn't have to clutter up the language with string operators.
- You can easily represent zero-terminated strings in any language able to create an array of characters.

However, zero-terminated strings also have disadvantages that mean they are not always the best choice for representing character string data:

- String functions that need to know the length of a string before working on the string data often aren't very efficient when operating on zero-terminated strings. The only reasonable way to compute the length of a zero-terminated string is to scan the string from the beginning to the end. The longer your strings are, the slower this function runs, so the zero-terminated string format isn't the best choice if you need to process long strings.
- Although it's a minor problem, you cannot easily represent the character code 0 (such as the NUL character in ASCII and Unicode) with the zero-terminated string format.
- Zero-terminated strings don't contain any information that tells you how long the string can grow beyond the terminating 0 byte. Therefore, some string functions, like concatenation, can only extend the length of an existing string variable and check for overflow if the caller explicitly passes the maximum length.

5.2.1.2 Length-Prefixed Strings

A second string format, *length-prefixed strings*, overcomes some of the problems with zero-terminated strings. Length-prefixed strings are common in languages like Pascal; they generally consist of a single byte that specifies the length of the string, followed by zero or more 8-bit character codes. In a length-prefixed scheme, the string "abc" consists of 4 bytes: the length byte (0x03), followed by a, b, and c.

Length-prefixed strings solve two of the problems associated with zero-terminated strings: they allow you to represent the NUL character, and string operations are more efficient. Another advantage to length-prefixed strings is that the length is usually located at position 0 in the string (if we view the string as an array of characters), so the first character of the string begins at index 1 in the array representation of the string. For many string functions, having a 1-based index into the character data is much more convenient than a 0-based index (which zero-terminated strings use).

The principal drawback of length-prefixed strings is that they are limited to a maximum of 255 characters in length (assuming a 1-byte length prefix). You can remove this limitation by using a 2- or 4-byte length value, but doing so increases the amount of overhead data from 1 to 2 or 4 bytes.

5.2.1.3 Seven-Bit Strings

The 7-bit string format is an interesting option that works for 7-bit encodings like ASCII. It uses the (normally unused) higher-order bit of the characters in the string to indicate the end of the string. All but the last character code in the string has its HO bit clear, and the last character in the string has its HO bit set.

This 7-bit string format has several disadvantages:

- You have to scan the entire string in order to determine the length of the string.
- You cannot have zero-length strings.
- Few languages provide literal string constants for 7-bit strings.
- You're limited to a maximum of 128 character codes, though this is fine when you're using plain ASCII.

However, a big advantage of 7-bit strings is that they don't require any overhead bytes to encode the length. Assembly language (using a macro to create literal string constants) is probably the best language to use when dealing with 7-bit strings. Because the benefit of 7-bit strings is that they're compact and assembly language programmers tend to worry most about compactness, this is a good match. Here's an HLA macro that converts a literal string constant to a 7-bit string:

```
#macro sbs( s );

    // Grab all but the last character of the string:

    (@substr( s, 0, @length(s) - 1) +

    // Concatenate the last character with its H0 bit set:

    char( uns8( char( @substr( s, @length(s) - 1, 1))) | $80 ) )

#endmacro

. . .
byte sbs( "Hello World" );
```

5.2.1.4 HLA Strings

As long as you're not too concerned about a few extra bytes of overhead per string, you can create a string format that combines the advantages of both length-prefixed and zero-terminated strings without their respective disadvantages. The High-Level Assembly language has done this with its native string format.⁸

The biggest drawback to the HLA character string format is the amount of overhead required for each string: 9 bytes per string,⁹ which can be significant, percentage-wise, if you're in a memory-constrained environment and you process many small strings.

The HLA string format uses a 4-byte length prefix, allowing character strings to be just over four billion characters long (obviously, this is far

8. Note that HLA is an assembly language, so it's perfectly possible—and easy—to support any reasonable string format. HLA's native string format is what it uses for literal string constants, and what most of the routines in the HLA standard library support.

9. Actually, because of memory alignment restrictions, there can be up to 12 bytes of overhead, depending on the string.

more than any practical HLA application will use). HLA also appends a 0 byte to the character string data. The additional 4 bytes of overhead contain the maximum legal length for that string. Having this extra field allows HLA string functions to check for string overflow, if necessary. In memory, HLA strings take the form shown in Figure 5-3.

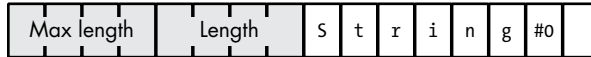


Figure 5-3: HLA string format

The 4 bytes immediately before the first character of the string contain the current string length. The 4 bytes preceding the current string length contain the maximum string length. Immediately following the character data is a 0 byte. Finally, HLA always ensures that the string data structure's length is a multiple of 4 bytes long (for performance reasons), so there may be up to 3 additional bytes of padding at the end of the object in memory. (Note that the string in Figure 5-3 requires only 1 byte of padding to ensure that the data structure is a multiple of 4 bytes in length.)

HLA string variables are pointers that contain the byte address of the first character in the string. To access the length fields, you load the value of the string pointer into a 32-bit register, then access the Length field at offset -4 from the base register and the MaxLength field at offset -8 from the base register. Here's an example:

```
static
    s:string := "Hello World";
    . . .
    mov( s, esi );           // Move the address of 'H' in "Hello World"
                           // into esi.
    mov( [esi-4], ecx );     // Puts length of string (11 for "Hello World")
                           // into ECX.
    . . .
    mov( s, esi );
    cmp( eax, [esi-8] );     // See if value in EAX exceeds the maximum
                           // string length.
    ja StringOverflow;
```

As read-only objects, HLA strings are compatible with zero-terminated strings. For example, if you have a function written in C that's expecting you to pass it a zero-terminated string, you can call that function and pass it an HLA string variable, like this:

```
someCFunc( hlaStringVar );
```

The only catch is that the C function must not make any changes to the string that would affect its length (because the C code won't update the Length field of the HLA string). Of course, you can always call a C `strlen()` function upon returning to update the length field yourself, but generally, it's best not to pass HLA strings to a function that modifies zero-terminated strings.

5.2.1.5 Descriptor-Based Strings

The string formats we've considered up to this point have kept the attribute information (that is, the lengths and terminating bytes) for a string in memory along with the character data. A slightly more flexible scheme is to maintain such information in a record structure, known as a *descriptor*, that also contains a pointer to the character data. Consider the following Pascal/Delphi data structure:

```
type
  dString :record
    curLength :integer;
    strData   :^char;
  end;
```

Note that this data structure does not hold the actual character data. Instead, the `strData` pointer contains the address of the first character of the string. The `curLength` field specifies the current length of the string. You could add any other fields you like to this record, like a maximum length field, though a maximum length isn't usually necessary because most string formats employing a descriptor are *dynamic* (as the next section will discuss). Most string formats employing a descriptor just maintain the `Length` field.

An interesting attribute of a descriptor-based string system is that the actual character data associated with a string could be part of a larger string. Because there are no length or terminating bytes within the actual character data, it's possible to have the character data for two strings overlap (see Figure 5-4).

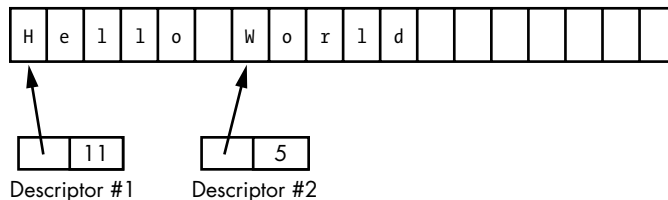


Figure 5-4: Overlapping strings using descriptors

In this example, there are two strings—"Hello World" and "World"—that overlap. This can save memory and make certain functions, like `substring()`, very efficient. Of course, when strings overlap like this, you can't modify the string data because that could wipe out part of some other string.

5.2.1.6 Java Strings

Java uses a descriptor-based string form. The actual `String` data type (that is, the structure/class that defines the internal representation of a Java string) is *opaque*, which means you really aren't supposed to know about or mess with it. It's a very bad idea to attempt to manipulate Java strings other than via the Java `String` API, because the Java standard has changed their internal representation on a couple of occasions.

For example, Java originally defined the `String` type as a descriptor with four items: a pointer to an array of 16-bit (original) Unicode characters (no extension beyond 16 bits), a count field, an offset field, and a hash code field. The offset and count fields allowed efficient substring operations, since all substrings into a larger string would share the same array of characters. Unfortunately, this format produced memory leaks in some degenerate cases, so Java's designers changed the format and eliminated these fields. If you had code that used the offset and count fields (again, a bad idea), your code was broken by this change.

Java also switched from the original Unicode 2-byte definition to UTF-16 encoding once it became apparent that 16-bit characters were insufficient. However, after a bit of research into a wide variety of Java programs on the internet, Oracle (Java's owner) discovered that most programs use only the Latin-1 character set (basically, ASCII). In Oracle's own words:

Data from different applications suggests that strings are a major component of Java heap usage and that most `java.lang.String` objects contain only Latin-1 characters. Such characters require only one byte of storage. As a result, half of the space in the internal character arrays of `java.lang.String` objects are not used. The compact strings feature, introduced in Java SE 9, reduces the memory footprint, and reduces garbage collection activity.

This change was largely transparent to Java users and their programs. Oracle added a new field to the `String` descriptor to specify whether the encoding was UTF-16 or Latin-1. Once again, if your programs depended on the internal representation, they broke.

Always assume that Java Strings are proper Unicode strings (typically using UTF-16 encoding). Java does not try to hide the ugliness of multiword characters. As a Java programmer, you must be aware of the difference between the number of characters, code points, and grapheme clusters in a string. Java provides functions—for example, `String.length()`, `String.codePointCount()`, and `BreakIterator.getCharacterInstance()`—to compute all these values for you, but your code must explicitly call them.

5.2.1.7 Swift Strings

Like Java, the Swift programming language uses Unicode characters in its strings. Swift 4.x and earlier used a UTF-16 encoding, which is native to macOS (on which Apple developed Swift); with Swift v5.0, Apple switched to UTF-8 as the native encoding for Swift strings. As with Java, Swift's `String` type is opaque, so you shouldn't attempt to mess with (or otherwise use) its internal representation.

5.2.1.8 C# Strings

The C# programming language uses UTF-16 encoding for characters in its strings. As with Java and Swift, C#'s `string` type is opaque and you shouldn't attempt to mess with (or otherwise use) its internal representation. That

being said, the Microsoft documentation does claim that C# strings are an array of (Unicode) characters.

5.2.1.9 Python Strings

The Python programming language originally used UCS-2 (original 16-bit Unicode, BMP-only) encoding for strings. Then Python was modified to support UTF-16 or UTF-32 encodings (the language was compiled in “narrow” or “wide” versions for 16- or 32-bit characters). Today, modern versions of Python use a special string format that tracks the characters in strings and stores them as ASCII, UTF-8, UTF-16, or UTF-32, based on the most compact representation. You can’t really access the internal string representation directly within Python, so the caveats of opaque types aren’t relevant.

5.2.2 Types of Strings: Static, Pseudo-Dynamic, and Dynamic

Based on the various string formats covered thus far, we can now define three string types according to when the system allocates storage for the string. There are static, pseudo-dynamic, and dynamic strings.

5.2.2.1 Static Strings

Pure *static strings* are those whose maximum size a programmer chooses when writing the program. Pascal strings and Delphi “short” strings fall into this category. Arrays of characters that you use to hold zero-terminated strings in C/C++ also fall into this category. Consider the following declaration in Pascal:

```
(* Pascal static string example *)  
  
var  pascalString :string(255); // Max length will always be 255 characters.
```

And here’s an example in C/C++:

```
// C/C++ static string example:  
  
char cString[256]; // Max length will always be 255 characters  
                  // (plus 0 byte).
```

While the program is running, there’s no way to increase the maximum sizes of these static strings. Nor is there any way to reduce the storage they will use; these string objects will consume 256 bytes at runtime, period. One advantage to pure static strings is that the compiler can determine their maximum length at compile time and implicitly pass this information to a string function so it can test for bounds violations at runtime.

5.2.2.2 Pseudo-Dynamic Strings

A pseudo-dynamic string is one whose length the system sets at runtime by calling a memory management function like `malloc()` to allocate storage for it. However, once the system allocates storage for the string, the maximum

length of the string is fixed. HLA strings generally fall into this category.¹⁰ An HLA programmer typically calls the `stralloc()` function to allocate storage for a string variable, after which that particular string object has a fixed length that cannot change.¹¹

5.2.2.3 Dynamic Strings

Dynamic string systems, which typically use a descriptor-based format, automatically allocate sufficient storage for a string object whenever you create a new string or otherwise do something that affects an existing string. Operations like string assignment and substring are relatively trivial in dynamic string systems—generally they copy only the string descriptor data, so these operations are fast. However, as noted in the section “Descriptor-Based Strings” on page 114, when using strings this way, you cannot store data back into a string object, because it could modify data that is part of other string objects in the system.

The solution to this problem is to use the copy-on-write technique. Whenever a string function needs to change characters in a dynamic string, the function first makes a copy of the string and then makes the necessary modifications to that copy. Research suggests that copy-on-write semantics can improve the performance of many typical applications, because operations like string assignment and substring extraction (which is just a partial string assignment) are far more common than the modification of character data within strings. The only drawback to this approach is that after several modifications to string data in memory, there may be sections of the string heap area that contain character data that’s no longer in use. To avoid a *memory leak*, dynamic string systems employing copy on write usually provide *garbage collection* code, which scans the string heap area looking for *stale* character data in order to recover that memory for other purposes. Unfortunately, depending on the algorithms in use, garbage collection can be quite slow.

5.2.3 Reference Counting for Strings

Consider the case where you have two string descriptors (or pointers) pointing at the same string data in memory. Clearly, you can’t *deallocate* (that is, reuse for a different purpose) the storage associated with one pointer while the program is still using the other pointer to access the same data. One common solution is to make the programmer responsible for keeping track of such details. Unfortunately, as applications become more complex, this approach often leads to dangling pointers, memory leaks, and other pointer-related problems in the software. A better solution is to allow the

10. Though, being assembly language, it’s possible to create static strings and pure dynamic strings in HLA as well.

11. Actually, you could call `strrealloc()` to change the size of an HLA string, but dynamic string systems generally do this automatically. Existing HLA string functions will not do this for you if they detect a string overflow.

programmer to deallocate the storage for the character data in the string and to have the actual deallocation process hold off until the programmer releases the last pointer referencing that data. To accomplish this, a string system can use reference counters, which track the pointers and their associated data.

A *reference counter* is an integer that counts the number of pointers that reference a string's character data in memory. Every time you assign the address of the string to some pointer, you increment the reference counter by 1. Likewise, whenever you wish to deallocate the storage associated with the character data for the string, you decrement the reference counter. Deallocation of the storage for the character data doesn't happen until the reference counter decrements to 0.

Reference counting works great when the language handles the details of string assignment automatically for you. If you try to implement reference counting manually, you must be sure to always increment the reference counter when you assign a string pointer to some other pointer variable. The best way to do this is to never assign pointers directly, but rather to handle all string assignments via some function (or macro) call that updates the reference counters in addition to copying the pointer data. If your code fails to update the reference counter properly, you'll wind up with dangling pointers or memory leaks.

5.2.4 Delphi Strings

Although Delphi provides a "short string" format that is compatible with the length-prefixed strings in earlier versions of Delphi, later versions of Delphi (4.0 and later) use dynamic strings. While this string format is unpublished (and, therefore, subject to change), indications are that Delphi's string format is very similar to HLA's. Delphi uses a zero-terminated sequence of characters with a leading string length and a reference counter (rather than a maximum length as HLA uses). Figure 5-5 shows the layout of a Delphi string in memory.

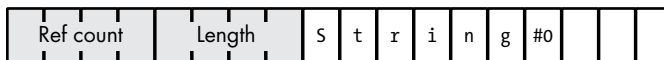


Figure 5-5: Delphi string data format

As with HLA, Delphi string variables are pointers that point to the first character of the actual string data. To access the length and reference counter fields, the Delphi string routines use a negative offset of -4 and -8 from the character data's base address. However, because this string format is not published, applications should never access the length or reference counter fields directly. Delphi provides a length function that extracts the string length for you, and there's really no need for your applications to access the reference counter field because the Delphi string functions maintain it automatically.

5.2.5 Custom String Formats

Typically, you'll use the string format your language provides, unless you have special requirements. If that's the case, you'll find that most languages provide user-defined data-structuring capabilities that enable you to create your own custom string formats.

Note that the language will probably insist on a single string format for literal string constants. However, you can usually write a short conversion function that will translate the literal strings in your language to whatever format you choose.

5.3 Character Set Data Types

Like strings, character set data types (or just *character sets*) are a composite data type built upon the character data type. A *character set* is a mathematical set of characters. Membership in a set is a binary relation: a character is either in the set or not, and you can't have multiple copies of the same character in a character set. Furthermore, the concept of sequence (whether one character comes before another, as in a string) is foreign to a character set. If two characters are members of a set, their order in the set is irrelevant.

Table 5-4 lists some common operations that applications perform on character sets.

Table 5-4: Common Character Set Functions

Function/operator	Description
Membership (in)	Checks to see if a character is a member of a character set (returns true/false).
Intersection	Returns the intersection of two character sets (that is, the set of characters that are members of both sets).
Union	Returns the union of two character sets (that is, all the characters that are members of either set or both sets).
Difference	Returns the difference of two sets (that is, those characters in one set that are not in the other).
Extraction	Extracts a single character from a set.
Subset	Returns true if one character set is a subset of another.
Proper subset	Returns true if one character set is a proper subset of another.
Superset	Returns true if one character set is a superset of another.
Proper superset	Returns true if one character set is a proper superset of another.
Equality	Returns true if one character set is equal to another.
Inequality	Returns true if one character set is not equal to another.

5.3.1 Powerset Representation of Character Sets

There are many different ways to represent character sets. Several languages implement them using an array of Boolean values (one Boolean value for each possible character code). Each Boolean value determines

whether its corresponding character is (true) or is not (false) a member of the character set. To conserve memory, most character set implementations allocate only a single bit for each character in the set; therefore, they consume 16 bytes (128 bits) of memory when supporting 128 characters, or 32 bytes (256 bits) when supporting up to 256 possible characters. This representation of a character set is known as a *powerset*.

The HLA language uses an array of 16 bytes to represent the 128 possible ASCII characters, which is organized in memory as shown in Figure 5-6.

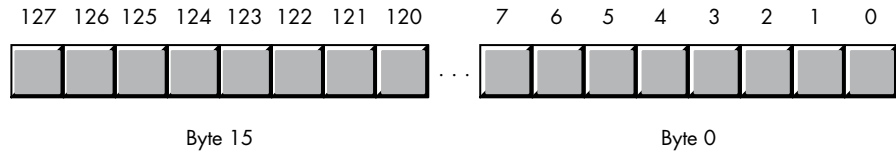


Figure 5-6: HLA character set representation

Bit 0 of byte 0 corresponds to ASCII code 0 (the NUL character). If this bit is 1, then the character set contains the NUL character; if this bit is 0, then the character set does not contain the NUL character. Likewise, bit 1 of byte 8 corresponds to ASCII code 65, an uppercase A. Bit 65 will contain a 1 if A is a current member of the character set, and 0 if it is not.

Pascal (for example, Delphi) uses a similar scheme to represent character sets. Delphi allows up to 256 characters in a character set, so Delphi character sets consume 256 bits (or 32 bytes) of memory.

While there are other ways to implement character sets, this bit vector (array) implementation makes it very easy to perform set operations like union, intersection, difference comparison, and membership tests.

5.3.2 List Representation of Character Sets

Sometimes a powerset bitmap just isn't the right representation for a character set. For example, if your sets are always very small (no more than three or four members), using 16 or 32 bytes to represent each of them can be overkill. In this case, you'd be better off using a character string to represent a list of characters.¹² If you rarely have more than a few characters in a set, scanning through a string to locate a particular character is probably efficient enough for most applications. Likewise, if your character set has a large number of possible characters, then the powerset representation could become huge (for example, implementing the original Unicode UCS-2 character set as a powerset would require 8,192 bytes of memory, even if there was only a single character in the set). In this situation, a list or character string representation could be more appropriate than a powerset, as you don't need to reserve memory for all possible members of the set (only those that are actually present).

12. Though it is up to you to ensure that the character string maintains set semantics (that is, you never allow duplicate characters in the string).

5.4 Designing Your Own Character Set

Very little is sacred about the ASCII, EBCDIC, and Unicode character sets. Their primary advantage is that they are international standards to which many systems adhere. If you stick with one of these standards, chances are good you'll be able to exchange information with other people, which is what these codes were designed for.

However, they were not designed to make various character computations easy. ASCII and EBCDIC were developed with now-antiquated hardware in mind—mechanical teletypewriters' keyboards and punched-card systems, respectively. Given that such equipment is found mainly in museums today, the layout of the codes in these character sets has almost no benefit in modern computer systems. If we could design our own character sets today, they'd be considerably different from ASCII or EBCDIC. They'd probably be based on modern keyboards (so they'd include codes for common keys, like LEFT ARROW, RIGHT ARROW, page up, and page down). They'd also be laid out to make various common computations a whole lot easier.

Although the ASCII and EBCDIC character sets are not going away any time soon, there's nothing stopping you from defining your own application-specific character set. Of course, such a set is, well, application-specific, and you won't be able to share text files containing characters encoded in your custom character set with applications that are ignorant of your private encoding. But it's fairly easy to translate between different character sets using a lookup table, so you can convert between your application's internal character set and an external character set (like ASCII) when performing I/O operations. Assuming you pick a reasonable encoding that makes your programs more efficient overall, the loss of efficiency during I/O can be worthwhile. But how do you choose an encoding?

The first question you have to ask yourself is, "How many characters do I want to support in my character set?" Obviously, the number of characters you choose will directly affect the size of your character data. An easy choice is 256 possible characters, because bytes are the most common primitive data type that software uses to represent character data. Keep in mind, however, that if you don't really need 256 characters, you probably shouldn't try to define that many in your character set. For example, if you can get by with 128, or even 64, characters in your custom character set, then "text files" you create with it will compress better. Likewise, data transmissions using it will be faster if you only have to transmit 6 or 7 bits for each character instead of 8. If you need more than 256 characters, you'll have to weigh the advantages and disadvantages of using multiple code pages, double-byte character sets, or 16-bit characters. And keep in mind that Unicode provides support for user-defined characters. So, if you need more than 256 characters in your character set, you might consider inserting it into Unicode to remain "somewhat standard" with the rest of the world.

In this section, we'll define a character set containing 128 characters using an 8-bit byte. For the most part, we'll simply rearrange the codes in the ASCII character set to make them more convenient for several calculations, and we'll rename a few of the control codes so they make sense on

modern systems instead of the old mainframes and teletypes for which they were created. We'll also add a few new characters beyond those defined by the ASCII standard. Again, the main purpose of this exercise is to make various computations more efficient, not create new characters. We'll call this the *HyCode* character set.

NOTE

This point bears repeating: the use of HyCode in this chapter is not an attempt to create some new character set standard. It's simply a demonstration of how you can create a custom, application-specific character set to improve your programs.

5.4.1 Designing an Efficient Character Set

We should think about several things when designing a new character set. For example, do we need to be able to represent strings of characters using an existing string format? This can influence the encoding of our strings—if you want to use function libraries that operate on zero-terminated strings, then you need to reserve encoding 0 in your custom character set for use as an end-of-string marker. Keep in mind, however, that a fair number of string functions won't work with your new character set, no matter what you do. Functions like `strcmp()` work only if you use the same representation for alphabetic characters as ASCII (or some other common character set). Therefore, you shouldn't feel hampered by the requirements of some particular string representation, because you're going to have to write many of your own string functions to process your custom characters anyway. The HyCode character set doesn't reserve code 0 for an end-of-string marker, and that's okay because zero-terminated strings are not very efficient.

If you look at programs that use character functions, you'll see that certain functions occur frequently, such as:

- Check a character to see if it is a digit.
- Convert a digit character to its numeric equivalent.
- Convert a numeric digit to its character equivalent.
- Check a character to see if it is alphabetic.
- Check a character to see if it is a lowercase character.
- Check a character to see if it is an uppercase character.
- Compare two characters (or strings) using a *case-insensitive* comparison.
- Sort a set of alphabetic strings (case-sensitive and case-insensitive sorting).
- Check a character to see if it is alphanumeric.
- Check a character to see if it is legal in an identifier.
- Check a character to see if it is a common arithmetic or logical operator.
- Check a character to see if it is a bracketing character (that is, one of `(, ), [, ], {, }, <, >`).
- Check a character to see if it is a punctuation character.

- Check a character to see if it is a *whitespace* character (such as a space, tab, or newline).
- Check a character to see if it is a cursor control character.
- Check a character to see if it is a scroll control key (such as PgUP, PgDN, HOME, and END).
- Check a character to see if it is a function key.

We'll design the HyCode character set to make these types of operations as efficient and easy as possible. One huge improvement we can make over the ASCII character set is to assign contiguous character codes to characters belonging to the same type, such as alphabetic characters and control characters, so we can do any of the preceding tests by using a pair of comparisons. For example, it would be nice if we could determine that a particular character is some sort of punctuation mark by comparing against two values that represent upper and lower bounds of the entire range of such characters, which we can't do in ASCII because the punctuation marks are spread throughout the character set. While it's not possible to satisfy every conceivable range comparison this way, we can design our character set to accommodate the most common tests with as few comparisons as possible.

5.4.2 Grouping the Character Codes for Numeric Digits

We can achieve the first three functions in the previous list by reserving the character codes 0 through 9 for the characters 0 through 9. First, by using a single unsigned comparison to check if a character code is less than or equal to 9, we can see if a character is a digit. Next, converting between characters and their numeric representations is trivial, because the character code and the numeric representation are one and the same.

5.4.3 Grouping Alphabetic Characters

The ASCII character set, though nowhere near as bad as EBCDIC, just isn't well designed for dealing with alphabetic character tests and operations. Here are some problems with ASCII that we'll solve with HyCode:

- The alphabetic characters lie in two disjoint ranges. Tests for an alphabetic character require four comparisons.
- The lowercase characters have ASCII codes that are greater than the uppercase characters. If we're going to do a case-sensitive comparison, it's more intuitive to treat lowercase characters as being less than uppercase characters.
- All lowercase characters have a greater value than any individual uppercase character. This leads to counterintuitive results, such as *a* being greater than *B*.

HyCode solves these problems in a couple of interesting ways. First, HyCode uses encodings \$4C through \$7F to represent the 52 alphabetic characters. Because HyCode uses only 128 character codes (\$00..\$7F), the

alphabetic codes consume the last 52 character codes. This means that we can test a character to see if it is alphabetic by comparing whether the code is greater than or equal to \$4C. In a high-level language, you'd write the comparison like this:

```
if( c >= 76) . . .
```

Or, if your compiler supports the HyCode character set, like this:

```
if( c >= 'a') . . .
```

In assembly language, you could use a pair of instructions like the following:

```
    cmp( al, 76 );
    jnae NotAlphabetic;

    // Execute these statements if it's alphabetic

NotAlphabetic:
```

HyCode interleaves the lowercase and uppercase characters (that is, the sequential encodings are for the characters *a*, *A*, *b*, *B*, *c*, *C*, and so on). This makes sorting and comparing strings very easy, regardless of whether you're doing a case-sensitive or case-insensitive search. The interleaving uses the LO bit of the character code to determine whether the character code is lowercase (LO bit is 0) or uppercase (LO bit is 1). HyCode uses the following encodings for alphabetic characters:

```
a:76, A:77, b:78, B:79, c:80, C:81, . . . y:124, Y:125, z:126, Z:127
```

Checking for an uppercase or lowercase alphabetic using HyCode is more work than checking whether a character is alphabetic, but in assembly it's still less work than the equivalent ASCII comparison. To test a character to see if it's a member of a single case, you need two comparisons—first to see if it's alphabetic, then to determine its case. In C/C++ you can use statements like the following:

```
if( (c >= 76) && (c & 1) )
{
    // execute this code if it's an uppercase character
}

if( (c >= 76) && !(c & 1) )
{
    // execute this code if it's a lowercase character
}
```

The subexpression `(c & 1)` evaluates true (1) if the LO bit of `c` is 1, meaning we have an uppercase character if `c` is alphabetic. Likewise, `!(c & 1)` evaluates true if the LO bit of `c` is 0, meaning we have a lowercase character. If you're working in 80x86 assembly language, you can test a character to see if it's uppercase or lowercase by using three machine instructions:

```
// Note: ROR(1, AL) maps lowercase to the range $26..$3F (38..63)
//       and uppercase to $A6..$BF (166..191). Note that all other characters
//       get mapped to smaller values within these ranges.
```

```
    ror( 1, al );
    cmp( al, $26 );
    jnae NotLower;    // Note: must be an unsigned branch!
```

```
    // Code that deals with a lowercase character.
```

NotLower:

```
// For uppercase, note that the ROR creates codes in the range $A8..$BF which
// are negative (8-bit) values. They also happen to be the *most* negative
// numbers that ROR will produce from the HyCode character set.
```

```
    ror( 1, al );
    cmp( al, $a6 );
    jge NotUpper;    // Note: must be a signed branch!
```

```
    // Code that deals with an uppercase character.
```

NotUpper:

Very few languages provide the equivalent of an `ror()` operation, and only a few allow you to (easily) treat character values as signed and unsigned within the same code sequence. Therefore, this sequence is probably limited to assembly language programs.

5.4.4 Comparing Alphabetic Characters

The HyCode grouping of alphabetic characters means that lexicographical ordering (“dictionary ordering”) is almost free. Sorting your strings by comparing the HyCode character values gives you lexicographical order, because HyCode defines the following relations on the alphabetic characters:

$$a < A < b < B < c < C < d < D < \dots < w < W < x < X < y < Y < z < Z$$

This is exactly the relationship you want for lexicographical ordering, and it's also the one most people would intuitively expect. To do a case-insensitive comparison, you simply mask out the LO bits (or force them both to 1) of the alphabetic characters.

To see the benefit of the HyCode character set when doing case-insensitive comparisons, let's first take a look at what the standard case-insensitive character comparison would look like in C/C++ for two ASCII characters:

```
if( toupper( c ) == toupper( d ))
{
    // do code that handles c==d using a case-insensitive comparison.
}
```

This code doesn't look too bad, but consider what the `toupper()` function (or, usually, macro) expands to:¹³

```
#define toupper(ch) ( (ch >= 'a' && ch <= 'z') ? ch & 0x5f : ch )
```

With this macro, you wind up with the following once the C preprocessor expands the previous if statement:

```
if
(
    ( (c >= 'a' && c <= 'z') ? c & 0x5f : c )
    == ( (d >= 'a' && d <= 'z') ? d & 0x5f : d )
)
{
    // do code that handles c==d using a case-insensitive comparison.
}
```

This expands to 80x86 code similar to this:

```
    // assume c is in cl and d is in dl.

    cmp( cl, 'a' );    // See if c is in the range 'a'..'z'
    jnb NotLower;
    cmp( cl, 'z' );
    ja NotLower;
    and( $5f, cl );    // Convert lowercase char in cl to uppercase.
NotLower:

    cmp( dl, 'a' );    // See if d is in the range 'a'..'z'
    jnb NotLower2;
    cmp( dl, 'z' );
    ja NotLower2;
    and( $5f, dl );    // Convert lowercase char in dl to uppercase.
NotLower2:

    cmp( cl, dl );     // Compare the (now uppercase if alphabetic)
                      // chars.
    jne NotEqual;     // Skip the code that handles c==d if they're
                      // not equal.
```

13. Actually, it's worse than this because most C standard libraries use lookup tables to map ranges of characters, but we'll ignore that issue here.

```
        // do code that handles c==d using a case-insensitive comparison.
NotEqual:
```

In HyCode, case-insensitive comparisons are much simpler. Here's what the HLA assembly code would look like:

```
// Check to see if CL is alphabetic. No need to check DL as the comparison
// will always fail if DL is nonalphabetic.
```

```
    cmp( cl, 76 );      // If CL < 76 ('a') then it's not alphabetic
    jb TestEqual;       // and there is no way the two chars are equal
                        // (even ignoring case).
```

```
    or( 1, cl );        // CL is alpha, force it to uppercase.
    or( 1, dl );        // DL may or may not be alpha. Force to
                        // uppercase if it is.
```

```
TestEqual:
    cmp( cl, dl );      // Compare the uppercase versions of the chars.
    jne NotEqual;       // Bail out if they're not equal.
```

```
TheyreEqual:
        // do code that handles c==d using a case-insensitive comparison.
```

```
NotEqual:
```

As you can see, the HyCode sequence uses half the instructions for a case-insensitive comparison of two characters.

5.4.5 Grouping Other Characters

Because alphabetic characters are at one end of the character code range and numeric characters are at the other, it takes two comparisons to check a character to see if it's alphanumeric (which is still better than the four comparisons necessary in ASCII). Here's the Pascal/Delphi code you'd use to see if a character is alphanumeric:

```
if( ch < chr(10) or ch >= chr(76) ) then . . .
```

Several programs (beyond compilers) need to efficiently process strings of characters that represent program identifiers. Most languages allow alphanumeric characters in identifiers, and, as you just saw, we can check a character to see if it's alphanumeric using only two comparisons.

Many languages also allow underscores within identifiers, and some languages, such as MASM, allow other characters like the "at" character (@) and dollar sign (\$) to appear within identifiers. Therefore, by assigning the underscore character the value 75, and by assigning the \$ and @ characters the respective codes 73 and 74, we can still test for an identifier character using only two comparisons.

For similar reasons, HyCode groups together the cursor control keys, the whitespace characters, the bracketing characters (parentheses, brackets, braces, and angle brackets), the arithmetic operators, the punctuation

characters, and so on. Table 5-5 lists the complete HyCode character set. If you study the numeric codes assigned to each character, you'll see that they allow for efficient computation of most of the character operations described earlier.

Table 5-5: The HyCode Character Set

Binary	Hex	Decimal	Character	Binary	Hex	Decimal	Character
0000_0000	00	0	0	0001_1110	1E	30	End
0000_0001	01	1	1	0001_1111	1F	31	Home
0000_0010	02	2	2	0010_0000	20	32	PgDn
0000_0011	03	3	3	0010_0001	21	33	PgUp
0000_0100	04	4	4	0010_0010	22	34	Left
0000_0101	05	5	5	0010_0011	23	35	Right
0000_0110	06	6	6	0010_0100	24	36	Up
0000_0111	07	7	7	0010_0101	25	37	Down/ linefeed
0000_1000	08	8	8	0010_0110	26	38	Nonbreaking space
0000_1001	09	9	9	0010_0111	27	39	Paragraph
0000_1010	0A	10	Keypad	0010_1000	28	40	Carriage return
0000_1011	0B	11	Cursor	0010_1001	29	41	Newline/ enter
0000_1100	0C	12	Function	0010_1010	2A	42	Tab
0000_1101	0D	13	Alt	0010_1011	2B	43	Space
0000_1110	0E	14	Control	0010_1100	2C	44	(
0000_1111	0F	15	Command	0010_1101	2D	45	)
0001_0000	10	16	Len	0010_1110	2E	46	[
0001_0001	11	17	Len128	0010_1111	2F	47	]
0001_0010	12	18	Bin128	0011_0000	30	48	{
0001_0011	13	19	Eos	0011_0001	31	49	}
0001_0100	14	20	Eof	0011_0010	32	50	<
0001_0101	15	21	Sentinel	0011_0011	33	51	>
0001_0110	16	22	Break/ interrupt	0011_0100	34	52	=
0001_0111	17	23	Escape/ cancel	0011_0101	35	53	^
0001_1000	18	24	Pause	0011_0110	36	54	
0001_1001	19	25	Bell	0011_0111	37	55	&
0001_1010	1A	26	Back tab	0011_1000	38	56	-
0001_1011	1B	27	Backspace	0011_1001	39	57	+

(continued)

Table 5-5: The HyCode Character Set (continued)

Binary	Hex	Decimal	Character	Binary	Hex	Decimal	Character
0001_1100	1C	28	Delete				
0001_1101	1D	29	Insert				
0011_1010	3A	58	*	0101_1101	5D	93	I
0011_1011	3B	59	/	0101_1110	5E	94	j
0011_1100	3C	60	%	0101_1111	5F	95	J
0011_1101	3D	61	~	0110_0000	60	96	k
0011_1110	3E	62	!	0110_0001	61	97	K
0011_1111	3F	63	?	0110_0010	62	98	l
0100_0000	40	64	,	0110_0011	63	99	L
0100_0001	41	65	.	0110_0100	64	100	m
0100_0010	42	66	:	0110_0101	65	101	M
0100_0011	43	67	;	0110_0110	66	102	n
0100_0100	44	68	"	0110_0111	67	103	N
0100_0101	45	69	'	0110_1000	68	104	o
0100_0110	46	70	`	0110_1001	69	105	O
0100_0111	47	71	\	0110_1010	6A	106	p
0100_1000	48	72	#	0110_1011	6B	107	P
0100_1001	49	73	\$	0110_1100	6C	108	q
0100_1010	4A	74	@	0110_1101	6D	109	Q
0100_1011	4B	75	_	0110_1110	6E	110	r
0100_1100	4C	76	a	0110_1111	6F	111	R
0100_1101	4D	77	A	0111_0000	70	112	s
0100_1110	4E	78	b	0111_0001	71	113	S
0100_1111	4F	79	B	0111_0010	72	114	t
0101_0000	50	80	c	0111_0011	73	115	T
0101_0001	51	81	C	0111_0100	74	116	u
0101_0010	52	82	d	0111_0101	75	117	U
0101_0011	53	83	D	0111_0110	76	118	v
0101_0100	54	84	e	0111_0111	77	119	V
0101_0101	55	85	E	0111_1000	78	120	w
0101_0110	56	86	f	0111_1001	79	121	W
0101_0111	57	87	F	0111_1010	7A	122	x
0101_1000	58	88	g	0111_1011	7B	123	X
0101_1001	59	89	G	0111_1100	7C	124	y
0101_1010	5A	90	h	0111_1101	7D	125	Y
0101_1011	5B	91	H	0111_1110	7E	126	z
0101_1100	5C	92	i	0111_1111	7F	127	Z

5.5 For More Information

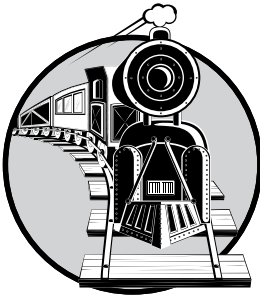
Hyde, Randall. “HLA Standard Library Reference Manual.” n.d. <http://www.plantation-productions.com/Webster/HighLevelAsm/HLADoc/> or <https://bit.ly/2W5G1or>.

IBM. “ASCII and EBCDIC Character Sets.” n.d. <https://ibm.co/33aPn3t>.

Unicode, Inc. “Unicode Technical Site.” Last updated March 4, 2020. <https://www.unicode.org/>.

6

MEMORY ORGANIZATION AND ACCESS



This chapter describes the basic components of a computer system: the CPU, memory, I/O, and the bus that connects them. We'll begin by discussing bus organization and memory organization. These two hardware components may have as large a performance impact on your software as the CPU's speed. Understanding memory performance characteristics, data locality, and cache operation can help you design software that runs as fast as possible.

6.1 The Basic System Components

The basic operational design of a computer system is called its *architecture*. John von Neumann, a pioneer in computer design, is credited with the principal architecture in use today. For example, the 80x86 family uses the *von Neumann architecture* (VNA). A typical VNA has three major components: the *central processing unit* (CPU), *memory*, and *input/output* (I/O), as shown in Figure 6-1.

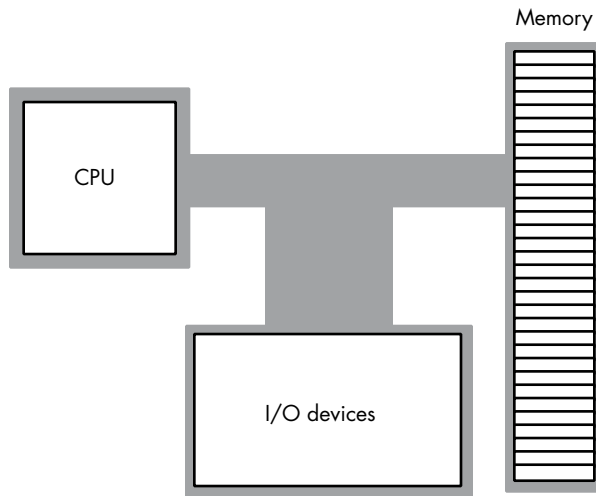


Figure 6-1: Typical von Neumann machine

In VNA machines, like the 80x86 systems, all computations occur within the CPU. Data and machine instructions reside in memory until the CPU requires them, at which point the system transfers the data into the CPU. To the CPU, most I/O devices look like memory; the major difference between them is that I/O devices are generally located in the outside world, whereas memory is located within the same machine.

6.1.1 The System Bus

The *system bus* connects the various components of a VNA machine. A *bus* is a collection of wires on which electrical signals pass between system components. Most CPUs have three major buses: the *data* bus, the *address* bus, and the *control* bus. These buses vary from processor to processor, but each bus carries comparable information on most CPUs. For example, the data buses on the Pentium and 80386 have different implementations, but both variants carry data between the processor, I/O, and memory.

6.1.1.1 The Data Bus

CPUs use the data bus to shuttle data between the various components in a computer system. The size of this bus varies widely among CPUs. Indeed, bus size (or *width*) is one of the main attributes that defines the “size” of the processor.

Most modern, general-purpose CPUs (such as those in PCs) employ a 32-bit-wide or, more commonly, 64-bit-wide data bus. Some processors use 8-bit or 16-bit data buses, and there may well be some CPUs with 128-bit data buses by the time you read this.

You’ll often hear the terms *8-, 16-, 32-, or 64-bit processor*. Processor size is determined by whichever value is smaller: the number of data lines on

the processor or the size of the largest general-purpose integer register. For example, older Intel 80x86 CPUs all have 64-bit buses but only 32-bit general-purpose integer registers, so they're classified as 32-bit processors. The AMD (and newer Intel) x86-64 processors support 64-bit integer registers and a 64-bit bus, so they're 64-bit processors.

Although the 80x86 family members with 8-, 16-, 32-, and 64-bit data buses can process data blocks up to the bit width of the bus, they can also access smaller memory units of 8, 16, or 32 bits. Therefore, anything you can do with a small data bus can be done with a larger data bus as well; the larger data bus, however, may access memory faster and can access larger chunks of data in one memory operation. You'll read about the exact nature of these memory accesses a little later in this chapter.

6.1.1.2 The Address Bus

The data bus on an 80x86 family processor transfers information between a particular memory location or I/O device and the CPU. *Which* memory location or I/O device is where the address bus comes in. The system designer assigns each memory location and I/O device a unique memory address. When the software wants to access a particular memory location or I/O device, it places the corresponding address on the address bus. Circuitry within the device checks the address and, if it matches, transfers data. All other memory locations ignore the request on the address bus.

With a single address bus line, a processor can access exactly two unique addresses: 0 and 1. With n address lines, the processor can access 2^n unique addresses (because there are 2^n unique values in an n -bit binary number). The number of bits on the address bus determines the *maximum* number of addressable memory and I/O locations. Early 80x86 processors, for example, provided only 20 lines on the address bus. Therefore, they could access only up to 1,048,576 (or 2^{20}) memory locations. Larger address buses can access more memory (see Table 6-1).

Table 6-1: 80x86 Addressing Capabilities

Processor	Address bus size	Maximum addressable memory
8088, 8086, 80186, 80188	20	1,048,576 (1MB)
80286, 80386sx	24	16,777,216 (16MB)
80386dx	32	4,294,976,296 (4GB)
80486, Pentium	32	4,294,976,296 (4GB)
Pentium Pro, II, III, IV	36	68,719,476,736 (64GB)
Core, i3, i5, i7, i9	≥ 40	≥1,099,511,627,776 (≥1TB)

Newer processors will support larger address buses. Many other processors (such as ARM and IA-64) already provide much larger addresses buses and, in fact, support addresses up to 64 bits in the software.

A 64-bit address range is truly infinite as far as memory is concerned. No one will ever put 2^{64} bytes of memory into a computer system and feel that they need more. Of course, people have made claims like this in the past. A few years ago, no one ever thought a computer would need 1GB of memory, yet computers with 64GB of memory (or more) are very common today. However, 2^{64} is effectively infinity for one simple reason—it's physically impossible to build that much memory based on estimates of the current size (about 2^{86} different elementary particles) of the universe. Unless you can attach 1 byte of memory to every elementary particle on the planet, you won't even come close to approaching 2^{64} bytes of memory on a given computer system. Then again, maybe we really will use whole planets as computer systems one day, as Douglas Adams predicted in *The Hitchhiker's Guide to the Galaxy*. Who knows?

While the newer 64-bit processors have an internal 64-bit address space, they rarely bring out 64 address lines on the chip. This is because pins are a precious commodity on large CPUs, and it doesn't make sense to bring out extra address pins that will never be used. Currently, 40- to 52-bit address buses are the upper limit. In the distant future, this may expand a bit, but it's hard to imagine the need for, or even possibility of, a physical 64-bit address bus.

On modern processors, CPU manufacturers are building memory controllers directly onto the CPU. Instead of having a traditional address and data bus to which you connect arbitrary memory devices, newer CPUs contain specialized buses intended to talk to very specific *dynamic random-access memory (DRAM)* modules. A typical CPU's memory controller connects to only a certain number of DRAM modules; thus, the maximum DRAM you can easily connect to a CPU is a function of the memory control built into the CPU rather than the size of the external address bus. This is why some older laptops have a 16MB or 32MB maximum memory limitation even though they have 64-bit CPUs.¹

6.1.1.3 The Control Bus

The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. To understand its importance, consider the data bus for a moment. The CPU uses the data bus to move data between itself and memory. The system uses two lines on the control bus, *read* and *write*, to determine the data flow direction (CPU to memory, or memory to CPU). So, when the CPU wants to write data to memory, it *asserts* (places a signal on) the write control line. When the CPU wants to read data from memory, it asserts the read control line.

Although the exact composition of the control bus varies among processors, some control lines—like the system clock lines, interrupt lines, status lines, and byte enable lines—are common to all processors. The byte

1. Technically, the laptop manufacturer could add a lot of external circuitry, including an external (to the CPU) memory controller, to overcome this limitation. However, such designs are expensive, so you rarely see them.

enable lines appear on the control bus of some CPUs that support byte-addressable memory. These control lines allow 16-, 32-, and 64-bit processors to deal with smaller chunks of data by communicating the size of the accompanying data. Additional details appear in the sections “16-Bit Data Buses” on page 138 and “32-Bit Data Buses” on page 140.

On the 80x86 family of processors, the control bus also contains a signal that helps distinguish between address spaces. The 80x86 family, unlike many other processors, provides two distinct address spaces: one for memory and one for I/O. However, it has only one physical address bus, shared between I/O and memory, so additional control lines decide which component the address is intended for. When these signals are active, the I/O devices use the address on the LO 16 bits of the address bus. When they’re inactive, the I/O devices ignore them, and the memory subsystem takes over at that point.

6.2 Physical Organization of Memory

A typical CPU addresses a maximum of 2^n different memory locations, where n is the number of bits on the address bus (most computer systems built around 80x86 family CPUs do not include the maximum addressable amount of memory). But what exactly is a memory location? The 80x86, as an example, supports *byte-addressable memory*. Therefore, the basic memory unit is a byte. With address buses containing 20, 24, 32, 36, or 40 address lines, the 80x86 processors can address 1MB, 16MB, 4GB, 64GB, or 1TB of memory, respectively. Some CPU families do not provide byte-addressable memory; instead, they commonly address memory only in double-word or even quad-word chunks. However, because of the vast amount of software that *assumes* memory is byte-addressable (such as all those C/C++ programs out there), even CPUs that don’t support byte-addressable memory in hardware still use byte addresses and simulate byte addressing in software. We’ll return to this topic shortly.

Think of memory as an array of bytes. The address of the first byte is 0 and the address of the last byte is $2^n - 1$. For a CPU with a 20-bit address bus, the following pseudo-Pascal array declaration is a good approximation of memory:

```
Memory: array [0..1048575] of byte; // 1MB address space (20 bits)
```

To execute the equivalent of the Pascal statement `Memory [125] := 0;` the CPU places the value 0 on the data bus, places the address 125 on the address bus, and asserts the write line on the control bus, as shown in Figure 6-2.

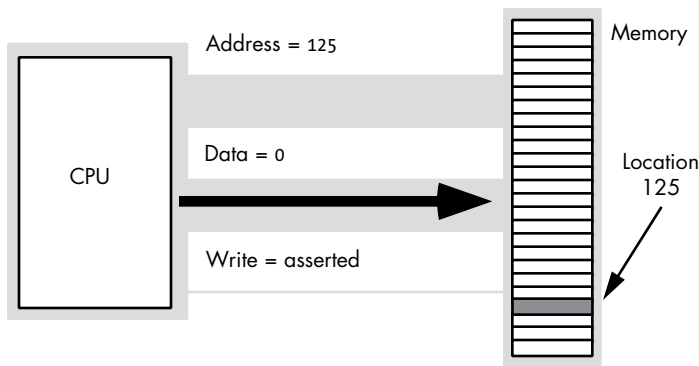


Figure 6-2: Memory write operation

To execute the equivalent of `CPU := Memory [125]`; the CPU places the address 125 on the address bus, asserts the read line on the control bus, and then reads the resulting data from the data bus (see Figure 6-3).

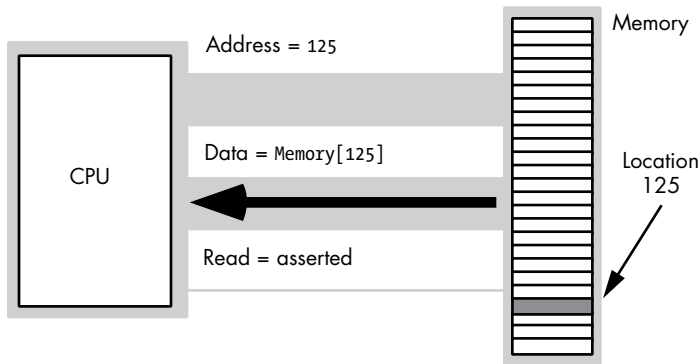


Figure 6-3: Memory read operation

This discussion applies *only* when the processor is accessing a single byte in memory. What happens when it accesses a word or a double word? Because memory consists of an array of bytes, how can we possibly deal with values larger than 8 bits?

Different computer systems have different solutions to this problem. The 80x86 family stores the LO byte of a word at the address specified and the HO byte at the next location. Therefore, a word consumes two consecutive memory addresses (as you would expect, because a word consists of 2 bytes). Similarly, a double word consumes four consecutive memory locations.

The address for a word or a double word is the address of its LO byte. The remaining bytes follow this LO byte, with the HO byte appearing at the address of the word plus 1 or the address of the double word plus 3 (see Figure 6-4).

It is quite possible for byte, word, and double-word values to overlap in memory. For example, in Figure 6-4, you could have a word variable

beginning at address 193, a byte variable at address 194, and a double-word value beginning at address 192. Bytes, words, and double words may begin at *any* valid address in memory. We'll soon see, however, that starting larger objects at an arbitrary address is not a good idea.

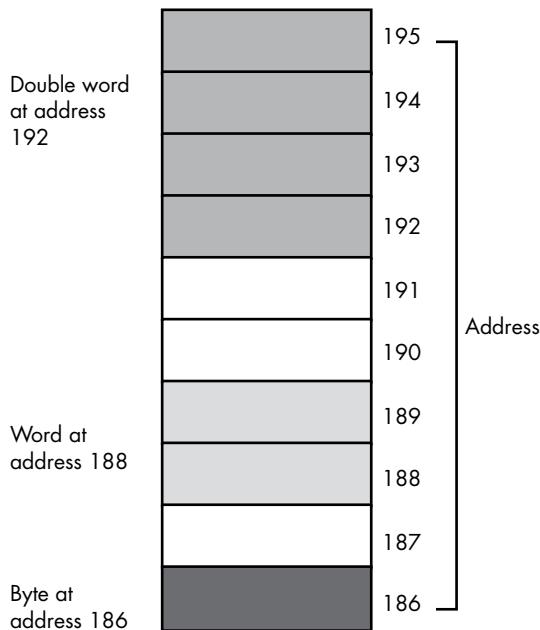


Figure 6-4: Byte, word, and double-word storage in memory (on an 80x86)

6.2.1 8-Bit Data Buses

A processor with an 8-bit bus (like the old 8088 CPU) can transfer 8 bits of data at a time. Because each memory address corresponds to an 8-bit byte, an 8-bit bus turns out to be the most convenient architecture (from the hardware perspective), as Figure 6-5 shows.

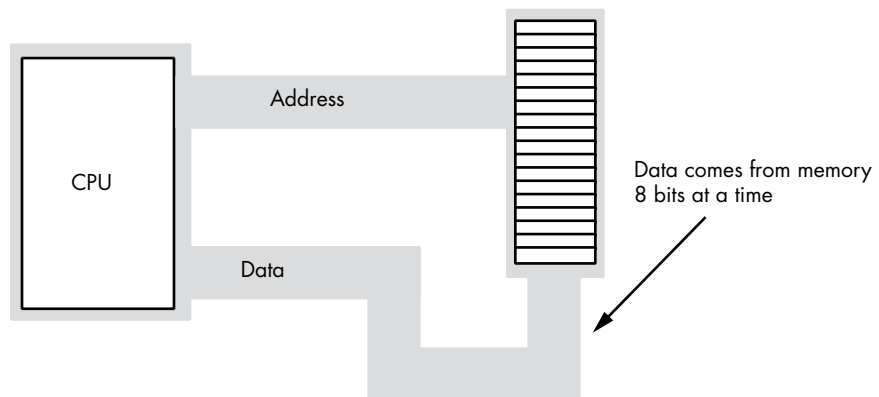


Figure 6-5: An 8-bit CPU <-> memory interface

The term *byte-addressable memory array* means that the CPU can address memory in chunks as small as a single byte. It also means that this is the *smallest* unit of memory you can access at once with the processor. That is, if the processor wants to access a 4-bit value, it must read 8 bits and then ignore the extra 4 bits.

Byte addressability *does not* imply that the CPU can access 8 bits starting at any arbitrary bit boundary. When you specify address 125 in memory, you get the entire 8 bits at that address—nothing less, nothing more. Addresses are integers; you cannot specify, for example, address 125.5 to fetch fewer than 8 bits or to fetch a byte straddling two byte addresses.

Although CPUs with an 8-bit data bus conveniently manipulate byte values, they can also manipulate word and double-word values. However, this requires multiple memory operations, because these processors can move only 8 bits of data at once. Loading a word requires two memory operations; loading a double word requires four memory operations.

6.2.2 16-Bit Data Buses

Some CPUs (such as the 8086, the 80286, and variants of the ARM processor family) have a 16-bit data bus. This allows these processors to access twice as much memory in the same amount of time as their 8-bit counterparts. These processors organize memory into two *banks*: an “even” bank and an “odd” bank (see Figure 6-6).

	Even	Odd	
Word 3	6	7	Numbers in cells represent the byte addresses
Word 2	4	5	
Word 1	2	3	
Word 0	0	1	

Figure 6-6: Byte addressing in word memory

Figure 6-7 illustrates the data bus connection to the CPU. In this figure, the data bus lines D0 through D7 transfer the LO byte of the word, while bus lines D8 through D15 transfer the HO byte of the word.

The 16-bit members of the 80x86 family can load a word from any arbitrary address. As mentioned earlier, the processor fetches the LO byte of the value from the address specified and the HO byte from the next consecutive address. However, this creates a subtle problem. What happens when you access a word that begins on an odd address? Suppose you want to read a word from location 125. The LO byte of the word comes from location 125 and the HO byte of the word comes from location 126. It turns out that there are actually *two* problems with this approach.

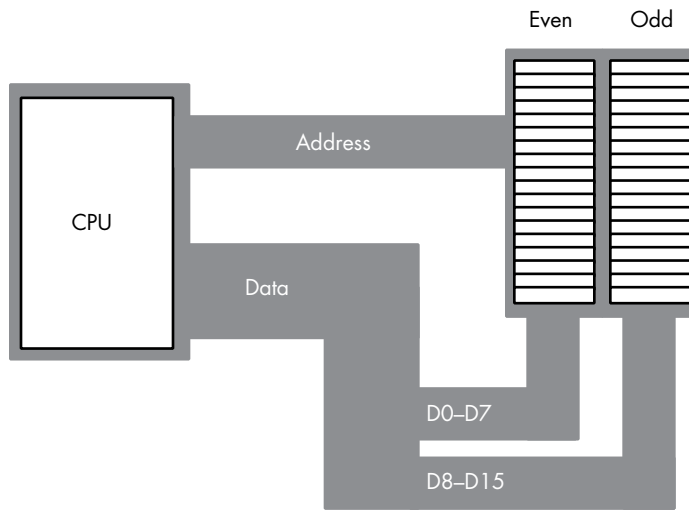


Figure 6-7: A 16-bit processor memory organization

As you can see in Figure 6-7, data bus lines 8 through 15 (the HO byte) connect to the odd bank, and data bus lines 0 through 7 (the LO byte) connect to the even bank. Accessing memory location 125 will transfer data to the CPU on lines D8 through D15 of the data bus, placing the data in the HO byte, yet we need this in the LO byte! Fortunately, the 80x86 CPUs automatically recognize and handle this situation.

The second problem is even more obscure. When accessing words, we're really accessing two separate bytes, each of which has its own byte address. So, what address appears on the address bus? The 16-bit 80x86 CPUs always place even addresses on the bus. Bytes at even addresses always appear on data lines D0 through D7, and bytes at odd addresses always appear on data lines D8 through D15. If you access a word at an even address, the CPU can bring in the entire 16-bit chunk in one memory operation. Likewise, if you access a single byte, the CPU activates the appropriate bank (using a byte-enable control line) and transfers that byte on the appropriate data lines for its address.

But what happens when the CPU accesses a word at an odd address, like the example given earlier? The CPU can't place address 125 on the address bus and read the 16 bits from memory. There are no odd addresses coming out of a 16-bit 80x86 CPU—they're always even. Therefore, if you try to put 125 on the address bus, 124 is what will actually appear there. Were you to read the 16 bits at this address, you would get the word at addresses 124 (LO byte) and 125 (HO byte)—not what you'd expect. Accessing a word at an odd address requires two memory operations (just as with the 8-bit bus on the 8088/80188). First, the CPU must read the byte at address 125, and then the byte at address 126. Second, it needs to swap the positions of these bytes internally because both entered the CPU on the wrong half of the data bus.

Fortunately, the 16-bit 80x86 CPUs hide these details from you. Your programs can access words at *any* address and the CPU will properly access and swap (if necessary) the data in memory. However, because of the two operations it requires, accessing words at odd addresses on a 16-bit processor is slower than accessing words at even addresses. By carefully arranging how you use memory, you can improve the speed of your programs on these CPUs.

6.2.3 32-Bit Data Buses

Accessing 32-bit quantities always takes at least two memory operations on the 16-bit processors. To access a 32-bit quantity at an odd address, a 16-bit processor may require three memory operations.

The 80x86 processors with a 32-bit data bus, such as the Pentium and Core processors, use four banks of memory connected to the 32-bit data bus (see Figure 6-8).

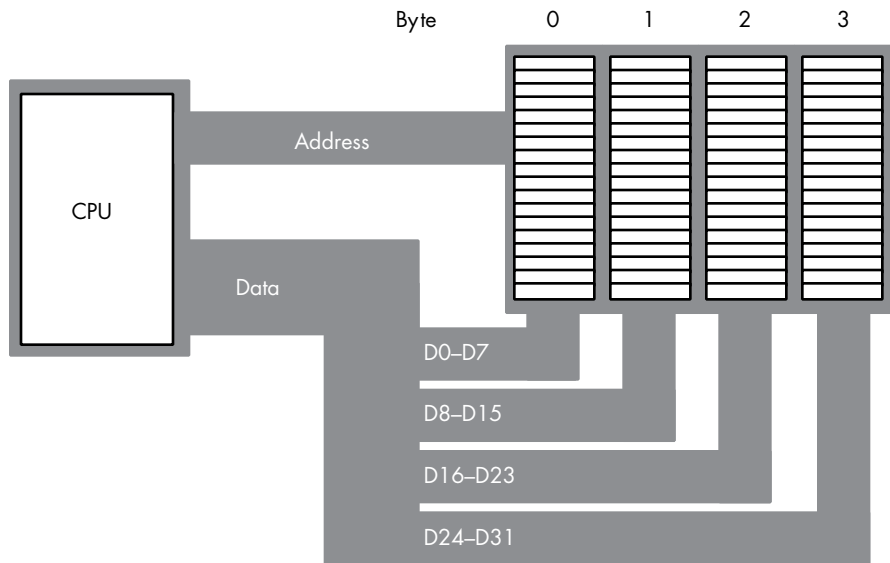


Figure 6-8: 32-bit processor memory interface

With a 32-bit memory interface, the 80x86 CPU can access any single byte with one memory operation. With a 16-bit memory interface, the address placed on the address bus is always an even number; and with a 32-bit memory interface, it's always some multiple of 4. Using various byte-enable control lines, the CPU can select which of the 4 bytes at that address the software wants to access. As with the 16-bit processor, the CPU will automatically rearrange bytes as necessary.

A 32-bit CPU can also access a word at most memory addresses using a single memory operation, though word accesses at certain addresses will take two memory operations (see Figure 6-9). This is the same problem we encountered with the 16-bit processor attempting to retrieve a word with an

odd address, except it occurs half as often—only when the address divided by 4 leaves a remainder of 3.

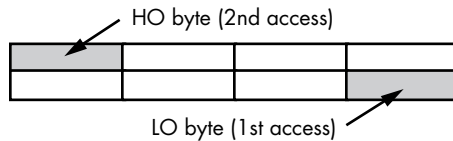


Figure 6-9: Accessing a word on a 32-bit processor at $(\text{address mod } 4) = 3$

A 32-bit CPU can access a double word in a single memory operation *only if* the address of that value is evenly divisible by 4. If not, the CPU may require two memory operations.

Once again, the 80x86 CPU handles all this automatically. However, there's a performance benefit to proper data alignment. Generally, the LO byte of word values should always be placed at even addresses, and the LO byte of double-word values should always be placed at addresses that are evenly divisible by 4.

6.2.4 64-Bit Data Buses

The Pentium and later processors, like Intel i-Series, provide a 64-bit data bus and special cache memory that reduces the impact of nonaligned data access. Although there may still be a penalty for accessing data at an inappropriate address, modern x86 CPUs suffer from the problem less frequently than the earlier CPUs. We'll look at the details in "Cache Memory" on page 151.

6.2.5 Small Accesses on Non-80x86 Processors

Although the 80x86 processor is not the only processor that will let you access a byte, word, or double-word object at an arbitrary byte address, most processors created in the past 30 years do *not* allow it. For example, the 68000 processor found in the original Apple Macintosh system would allow you to access a byte at any address, but raised an exception if you attempted to access a word at an odd address.² Many processors require that you access an object at an address that is a multiple of the object's size, or they'll raise an exception.

Most RISC processors, including those found in modern smartphones and tablets (typically ARM processors), do not allow you to access byte and word objects at all. Most RISC CPUs require that all data accesses be the same size as the data bus (or general-purpose integer register size, whichever is smaller). This is generally a double-word (32-bit) or quad-word (64-bit) access. If you want to access bytes or words on such a machine, you have to treat them as packed fields and use the shift and mask techniques to

2. 680x0 series processors starting with the 68020, found in later Macintosh systems, corrected this issue and allowed data access of words and double words at arbitrary addresses.

extract or insert byte and word data in a double word. Although it's nearly impossible to avoid byte accesses in software that does any character and string processing, if you expect your software to run efficiently on various modern RISC CPUs, you should avoid word data types (and the performance penalty for accessing them) in favor of double words.

6.3 Big-Endian vs. Little-Endian Organization

Earlier, you read that the 80x86 CPU family stores the LO byte of a word or double-word value at a particular address in memory and the successive HO bytes at successively higher addresses. Now we'll look in more depth at how different processors store multibyte objects in byte-addressable memory.

Almost every CPU whose "bit size" is some power of 2 (8, 16, 32, 64, and so on) numbers the bits and nibbles as shown in the previous chapters. There are some exceptions, but they are rare, and most of the time they represent a notational change, not a functional change (meaning you can safely ignore the difference). Once you start dealing with objects larger than 8 bits, however, things become more complicated. Different CPUs organize the bytes in a multibyte object differently.

Consider the layout of the bytes in a double word on an 80x86 CPU (see Figure 6-10). The LO byte, which contributes the smallest component of a binary number, sits in bit positions 0 through 7 and appears at the lowest address in memory. It seems reasonable that the bits that contribute the least would be located at the lowest address in memory.

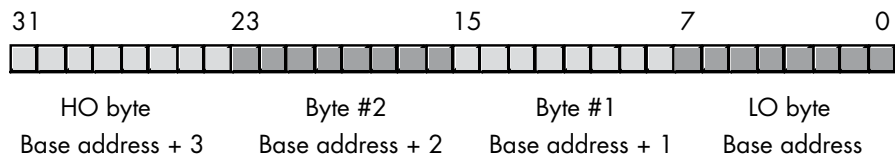


Figure 6-10: Byte layout in a double word on the 80x86 processor

This is not the only possible organization, however. Some CPUs reverse the memory addresses of all the bytes in a double word, using the organization shown in Figure 6-11.

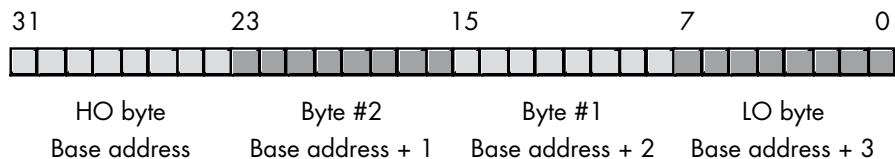


Figure 6-11: Alternate byte layout in a double word

The original Apple Macintosh (68000 and PowerPC) and most non-80x86 Unix boxes use the data organization shown in Figure 6-11. Even on 80x86 systems, certain protocols (such as network transmissions) specify this data organization. Therefore, this isn't some rare and esoteric

convention; it's quite common, and not something you can ignore if you work on PCs.

The byte organization that Intel uses is whimsically known as the *little-endian byte organization*. The alternate form is known as *big-endian byte organization*.

NOTE

These terms come from Jonathan Swift's Gulliver's Travels; the Lilliputians were arguing over whether one should open an egg by cracking it on the little end or the big end—a parody of the arguments the Catholics and Protestants were having over their respective doctrines when Swift was writing.

The time for arguing over which format is superior was back before there were several different CPUs created using different *endianness*. Today, that argument is irrelevant. Regardless of which format is better or worse, we have to deal with the fact that different CPUs sport different endianness, and we have to take care when writing software if we want our programs to run on both types of processors.

We encounter the big-endian versus little-endian problem when we try to pass binary data between two computers. For example, the double-word binary representation of 256 on a little-endian machine has the following byte values:

L0 byte:	0
Byte #1:	1
Byte #2:	0
H0 byte:	0

If you assemble these 4 bytes on a little-endian machine, their layout takes this form:

Byte:	3	2	1	0	
256:	0	0	1	0	(each digit represents an 8-bit value)

On a big-endian machine, however, the layout takes the following form:

Byte:	3	2	1	0	
256:	0	1	0	0	(each digit represents an 8-bit value)

This means that if you take a 32-bit value from one of these machines and attempt to use it on the other machine (with a different endianness), you won't get correct results. For example, if you take a big-endian version of the value 256 and interpret it as little-endian, you'll discover that it has a 1 in bit position 16, and a little-endian machine will think that the value is actually 65,536 (that is, %1_0000_0000_0000_0000).

When you're exchanging data between two different machines, the best solution is to convert your values to some canonical form and then convert the canonical form back to the local format if the local and canonical formats are not the same. Exactly what constitutes a "canonical" format depends, usually, on the transmission medium. For example, when you are

transmitting data across networks, the canonical form is usually big-endian because TCP/IP and some other network protocols use the big-endian format. When you're transmitting data across the Universal Serial Bus (USB), the canonical format is little-endian. Of course, if you control the software on both ends, the choice of canonical form is arbitrary; still, you should attempt to use the appropriate form for the transmission medium to avoid confusion down the road.

To convert between the endian forms, you must do a *mirror-image swap* of the bytes in the object: first swap the bytes at opposite ends of the binary number, and then work your way toward the middle of the object, swapping pairs of bytes as you go along. For example, to convert between the big-endian and little-endian format within a double word, you'd first swap bytes 0 and 3, then you'd swap bytes 1 and 2 (see Figure 6-12).

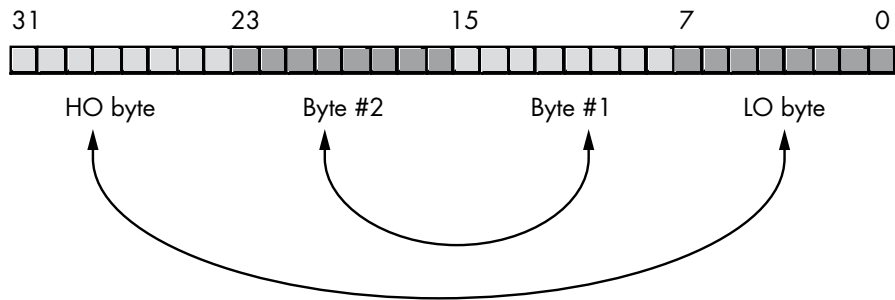


Figure 6-12: Endian conversion in a double word

For word values, all you need to do is swap the HO and LO bytes to change the endianness. For quad-word values, you need to swap bytes 0 and 7, 1 and 6, 2 and 5, and 3 and 4. Because very little software deals with 128-bit integers, you probably won't need to worry about long-word endianness conversion, but the concept is the same if you do.

Note that the endianness conversion process is *reflexive*; that is, the same algorithm that converts big-endian to little-endian also converts little-endian to big-endian. If you run the algorithm twice, you wind up with the data in the original format.

Even if you're not writing software that exchanges data between two computers, the issue of endianness may arise. Some programs assemble larger objects from discrete bytes by assigning those bytes to specific positions within the larger value. If the software puts the LO byte into bit positions 0 through 7 (little-endian format) on a big-endian machine, the program will not produce correct results. Therefore, if the software needs to run on different CPUs that have different byte organizations, it will have to determine the endianness of the machine it's running on and adjust how it assembles larger objects from bytes accordingly.

To illustrate how to build larger objects from discrete bytes, we'll start with a short example that demonstrates how you could assemble a 32-bit object from 4 individual bytes. The most common way to do this is

to create a *discriminant union* structure that contains a 32-bit object and a 4-byte array.

NOTE

Many languages, but not all, support the discriminant union data type. For example, in Pascal, you would instead use a case variant record. See your language reference manual for details.

Unions are similar to records or structures except the compiler allocates the storage for each field of the union at the same address in memory. Consider the following two declarations from the C programming language:

```
struct
{
    short unsigned i;    // Assume shorts require 16 bits.
    short unsigned u;
    long unsigned r;     // Assume longs require 32 bits.
} RECORDvar;

union
{
    short unsigned i;
    short unsigned u;
    long unsigned r;
} UNIONvar;
```

As Figure 6-13 shows, the RECORDvar object consumes 8 bytes in memory, and the fields do not share their memory with any other fields (that is, each field starts at a different offset from the base address of the record). The UNIONvar object, on the other hand, overlays all the fields in the union in the same memory locations. Therefore, writing a value to the *i* field of the union also overwrites the value of the *u* field as well as 2 bytes of the *r* field (whether they are the LO or HO bytes depends entirely on the endianness of the CPU).

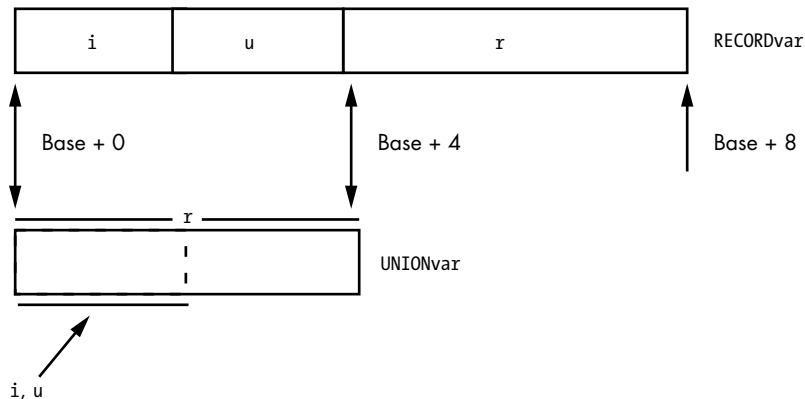


Figure 6-13: Layout of a union versus a record (struct) in memory

In the C programming language, you can use this behavior to access the individual bytes of a 32-bit object. Consider the following union declaration in C:

```
union
{
    unsigned long bits32; /* This assumes that C uses 32 bits for
                           unsigned long */
    unsigned char bytes[4];
} theValue;
```

This creates the data type shown in Figure 6-14 on a little-endian machine, and the structure shown in Figure 6-15 on a big-endian machine.

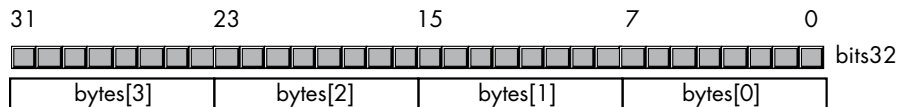


Figure 6-14: A C union on a little-endian machine

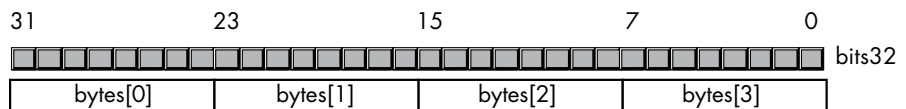


Figure 6-15: A C union on a big-endian machine

To assemble a 32-bit object from 4 discrete bytes on a little-endian machine, you'd use code like the following:

```
theValue.bytes[0] = byte0;
theValue.bytes[1] = byte1;
theValue.bytes[2] = byte2;
theValue.bytes[3] = byte3;
```

This code functions properly because C allocates the first byte of an array at the lowest address in memory (corresponding to bits 0..7 in the theValue.bits32 object on a little-endian machine); the second byte of the array follows (bits 8..15), then the third (bits 16..23), and finally the HO byte (occupying the highest address in memory, corresponding to bits 24..31).

However, on a big-endian machine, this code won't work properly because theValue.bytes[0] corresponds to bits 24 through 31 of the 32-bit value rather than bits 0 through 7. To assemble this 32-bit value properly on a big-endian system, you'd need to use code like the following:

```
theValue.bytes[0] = byte3;
theValue.bytes[1] = byte2;
theValue.bytes[2] = byte1;
theValue.bytes[3] = byte0;
```

But how do you determine if your code is running on a little-endian or big-endian machine? This is actually a simple task. Consider the following C code:

```
theValue.bytes[0] = 0;
theValue.bytes[1] = 1;
theValue.bytes[2] = 0;
theValue.bytes[3] = 0;
isLittleEndian = theValue.bits32 == 256;
```

On a big-endian machine, this code sequence will store the value 1 into bit 16, producing a 32-bit value that is definitely not equal to 256, whereas on a little-endian machine this code will store the value 1 into bit 8, producing a 32-bit value equal to 256. Therefore, you can test the `isLittleEndian` variable to determine whether the current machine is little-endian (true) or big-endian (false).

6.4 The System Clock

Although modern computers are quite fast and getting faster all the time, they still require time to accomplish even the smallest tasks. On von Neumann machines, most operations are *serialized*, which means that the computer executes commands in a prescribed order.³ It wouldn't do, in the following code sequence, to execute the Pascal statement `I := I * 5 + 2;` before the statement `I := J;` finishes:

```
I := J;
I := I * 5 + 2;
```

These operations do not occur instantaneously. Moving a copy of `J` into `I` takes a certain amount of time. Likewise, multiplying `I` by 5 and then adding 2 and storing the result back into `I` takes time.

To execute statements in the proper order, the processor relies on the *system clock*, which serves as the timing standard within the system. To understand why certain operations take longer than others, you must first understand how the system clock functions.

The system clock is an electrical signal on the control bus that alternates between 0 and 1 periodically (see Figure 6-16). All activity within the CPU is synchronized with the edges (rising or falling) of this clock signal.

3. Note that modern CPUs support *out-of-order execution* whereby the CPU starts the execution of later instructions before earlier instructions finish execution. However, the CPUs usually attempt to retain the same semantics as in-order execution.

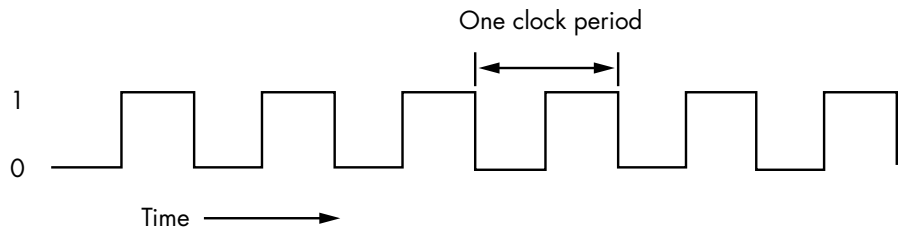


Figure 6-16: The system clock

The rate at which the system clock alternates between 0 and 1 is the *system clock frequency*, and the time it takes for the system clock to switch from 0 to 1 and back to 0 is the *clock period* or *clock cycle*. On most modern systems, the system clock frequency exceeds several billion cycles per second. A typical Pentium IV chip, circa 2004, runs at speeds of three billion cycles per second or faster. *Hertz (Hz)* is the unit corresponding to one cycle per second, so the aforementioned Pentium chip runs at between 3,000 and 4,000 million hertz, or 3,000 to 4,000 megahertz (MHz), or 3 to 4 gigahertz (GHz, or one billion cycles per second). Typical frequencies for 80x86 parts range from 5 MHz up to several gigahertz and beyond.

The clock period is the reciprocal of the clock frequency. For example, a 1 MHz (MHz or one million cycles per second) clock would have a clock period of 1 microsecond (one millionth of a second, μs ⁴). A CPU running at 1 GHz would have a clock period of one nanosecond (ns), or one billionth of a second. Clock periods are usually expressed in microseconds or nanoseconds.

To ensure synchronization, most CPUs start an operation on either the *falling edge* (when the clock goes from 1 to 0) or the *rising edge* (when the clock goes from 0 to 1). The system clock spends most of its time at either 0 or 1 and very little time switching between the two. Therefore, a clock edge is the perfect synchronization point.

Because all CPU operations are synchronized with the clock, the CPU cannot perform tasks any faster than the clock runs. However, just because a CPU is running at some clock frequency doesn't mean that it executes that many operations each second. Many operations take multiple clock cycles to complete, so the CPU often performs operations at a significantly slower rate.

6.4.1 Memory Access and the System Clock

Memory access is an operation that is synchronized with the system clock; that is, memory access occurs no more than once every clock cycle. On some older processors, it takes several clock cycles to access a memory location. The *memory access time* is the number of clock cycles between a

4. Often written as *us* when the Greek mu character is not available.

memory request (read or write) and when the memory operation completes. This is an important value, because longer memory access times result in lower performance.

Modern CPUs are much faster than memory devices, so systems built around these CPUs often use a second clock, the *bus clock*, which is some fraction of the CPU speed. For example, typical processors in the 100 MHz to 4 GHz range can use 1600 MHz, 800 MHz, 500 MHz, 400 MHz, 133 MHz, 100 MHz, or 66 MHz bus clocks (a given CPU generally supports several different bus speeds, and the exact range it supports depends upon that CPU).

When reading from memory, the memory access time is the time between when the CPU places an address on the address bus and the time when the CPU takes the data off the data bus. On typical 80x86 CPUs with a one-cycle memory access time, the timing of a read operation looks something like Figure 6-17. The timing of writing data to memory is similar (see Figure 6-18).

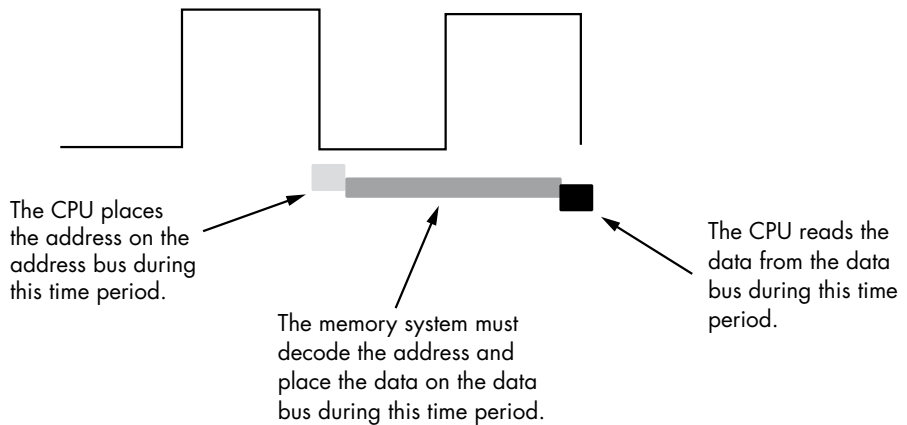


Figure 6-17: A typical memory read cycle



Figure 6-18: A typical memory write cycle

The CPU doesn't wait for memory. The access time is specified by the bus clock frequency. If the memory subsystem doesn't work fast enough to keep up with the CPU's expected access time, the CPU will read garbage data on a memory read operation and will not properly store the data on a memory write. This will surely cause the system to fail.

Memory devices have various ratings, but the two major ones are capacity and speed. Typical dynamic RAM (random access memory) devices have capacities of 16GB (or more) and speeds of 0.1 to 100 ns. A typical 4 GHz Intel system uses 1600 MHz (1.6 GHz, or 0.625 ns) memory devices.

Now, I just said that the memory speed must match the bus speed or the system will fail. At 4 GHz the clock period is roughly 0.25 ns. So how can a system designer get away with using 0.625 ns memory? The answer is *wait states*.

6.4.2 Wait States

A wait state is an extra clock cycle that gives a device additional time to respond to the CPU. For example, a 100 MHz Pentium system has a 10 ns clock period, implying that you need 10 ns memory. In fact, you need even faster memory devices because in many computer systems there's additional decoding and buffering logic between the CPU and memory, and this circuitry introduces its own delays. In Figure 6-19, you can see that buffering and decoding costs the system an additional 10 ns. If the CPU needs the data back in 10 ns, the memory must respond in 0 ns (which is impossible).

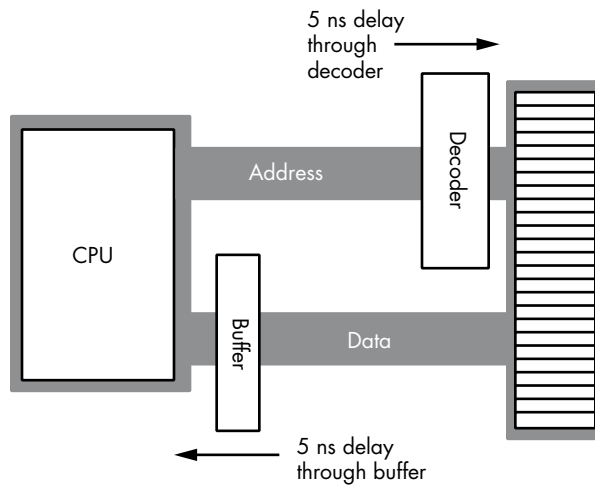


Figure 6-19: Decoding and buffer delays

If cost-effective memory won't work with a fast processor, how do companies manage to sell fast PCs? One part of the answer is the wait state. For example, if you have a 100 MHz processor with a memory cycle time of 10 ns and you lose 2 ns to buffering and decoding, you'll need 8 ns memory. What if your system can only support 20 ns memory, though? By adding wait states to extend the memory cycle to 20 ns, you can solve this problem.

Almost every general-purpose CPU in existence provides a pin (whose signal appears on the control bus) that allows you to insert wait states. If necessary, the memory address decoding circuitry asserts this signal to give the memory sufficient access time (see Figure 6-20).

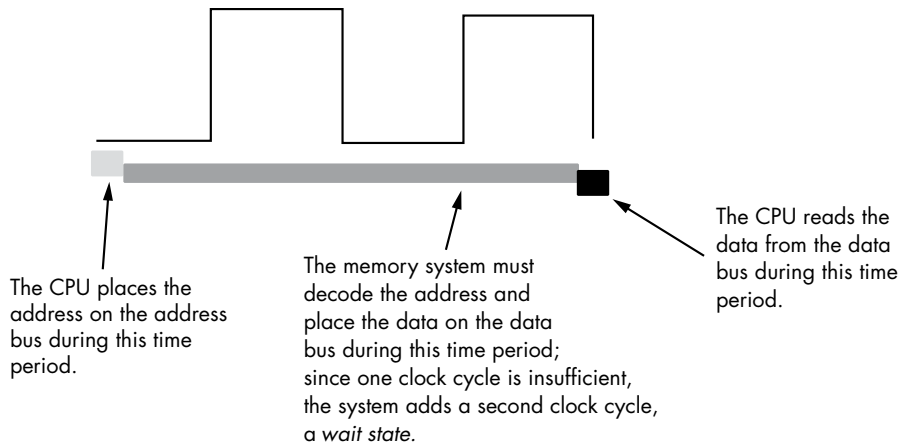


Figure 6-20: Inserting a wait state into a memory read operation

From the system performance point of view, wait states are *not* a good thing. As long as the CPU is waiting for data from memory, it can't operate on that data. Adding a wait state typically *doubles* (or worse, on some systems) the amount of time required to access memory. Running with a wait state on every memory access is almost like cutting the processor clock frequency in half. You'll get less work done in the same amount of time.

However, we're not doomed to slow execution because of added wait states. There are several tricks hardware designers can employ to achieve zero wait states *most* of the time. The most common is the use of *cache* (pronounced "cash") memory.

6.4.3 Cache Memory

A typical program tends to access the same memory locations repeatedly (known as *temporal locality of reference*), and to access adjacent memory locations (*spatial locality of reference*). Both forms of locality occur in the following Pascal code segment:

```
for i := 0 to 10 do
    A [i] := 0;
```

There are two occurrences each of spatial and temporal locality of reference within this loop. Let's consider the obvious ones first.

In this Pascal code, the program references the variable *i* several times. The for loop compares *i* against 10 to see if the loop is complete. It also increments *i* by 1 at the bottom of the loop. The assignment statement also uses *i* as an array index. This shows temporal locality of reference in action.

The loop itself zeros out the elements of array *A* by writing a 0 to the first location in *A*, then to the second location in *A*, and so on. Because Pascal stores the elements of *A* in consecutive memory locations, each loop iteration accesses adjacent memory locations. This shows spatial locality of reference.

What about the second occurrences of temporal and spatial locality? Machine instructions also reside in memory, and the CPU fetches these instructions sequentially from memory and executes them repeatedly, once for each loop iteration.

If you look at the execution profile of a typical program, you'll probably discover that the program executes less than half the statements. Generally, a program might use only 10 to 20 percent of the memory allotted to it. At any given time, a 1MB program might access only 4KB to 8KB of data and code. So, if you paid an outrageous sum of money for expensive zero-wait-state RAM, you'd be using only a tiny fraction of it at any given time. Wouldn't it be nice if you could buy a small amount of fast RAM and dynamically reassign its addresses as the program executes? This is exactly what cache memory does for you.

Cache memory is a small amount of very fast memory that sits between the CPU and main memory. Unlike in normal memory, the bytes within a cache do not have fixed addresses. Cache memory can dynamically reassign addresses, which allows the system to keep recently accessed values in the cache. Addresses that the CPU has never accessed, or hasn't accessed in some time, remain in main (slow) memory. Because most memory accesses are to recently accessed variables (or to locations near a recently accessed location), the data generally appears in cache memory.

A *cache hit* occurs whenever the CPU accesses memory and finds the data in the cache. In such a case, the CPU can usually access data with zero wait states. A *cache miss* occurs if the data cannot be found in the cache. In that case, the CPU has to read the data from main memory, incurring a performance loss. To take advantage of temporal locality of reference, the CPU copies data into the cache whenever it accesses an address that's not present in the cache. Because the system will likely access that address shortly, it can save wait states on future accesses by having that data in the cache.

Cache memory does not eliminate the need for wait states. Although a program may spend considerable time executing code in one area of memory, eventually it will call a procedure or wander off to some section of code outside cache memory. When that happens, the CPU has to go to main memory to fetch the data. Because main memory is slow, this will require the insertion of wait states. However, once the CPU accesses the data, it will be available in the cache for future use.

We've discussed how cache memory handles the temporal aspects of memory access, but not the spatial aspects. Caching memory locations *when you access them* won't speed up the program if you constantly access consecutive locations that you've never accessed before. To solve this problem, when a cache miss occurs, most caching systems will read several consecutive bytes of main memory (which engineers call a *cache line*). For example, 80x86 CPUs read between 16 and 64 bytes upon a cache miss. Most memory chips available today have special modes that let you quickly access several

consecutive memory locations on the chip. The cache exploits this capability to reduce the average number of wait states needed to access sequential memory locations. Although reading 16 bytes on each cache miss is expensive if you access only a few bytes in the corresponding cache line, cache memory systems work quite well in the average case.

The ratio of cache hits to misses increases with the size (in bytes) of the cache memory subsystem. The 80486 CPU, for example, has 8,192 bytes of on-chip cache. Intel claims to get an 80 to 95 percent hit rate with this cache (meaning 80 to 95 percent of the time the CPU finds the data in the cache). This sounds very impressive, but let's play around with the numbers a little bit. Suppose we pick the 80 percent figure. This means that one out of every five memory accesses, on average, will not be in the cache. If you have a 50 MHz processor (20 ns period) and a 90 ns memory access time, four out of five memory accesses require only 20 ns (one clock cycle) because they are in the cache, and the fifth will require about four wait states (20 ns for a normal memory access plus 80 additional ns, or four wait states, to get at least 90 ns). However, the cache always reads 16 consecutive bytes (4 double words) from memory. Most 80486-era memory subsystems let you read consecutive addresses in about 40 ns after accessing the first location. Therefore, the 80486 will require an additional six clock cycles to read the remaining 3 double words, for a total of 220 ns. This corresponds to 11 clock cycles (at 20 ns each), which is one normal memory cycle plus 10 wait states.

Altogether, the system will require 15 clock cycles to access five memory locations, or 3 clock cycles per access, on average. That's equivalent to two wait states added to every memory access. Doesn't sound so impressive, does it? It gets even worse as you move up to faster processors and the difference in speed between the CPU and memory increases.

To improve the hit ratio, you can add more cache memory. Alas, you can't pull an Intel i9 chip apart and solder more cache onto the chip. However, modern Intel CPUs have a significantly larger cache than the 80486 and operate with fewer average wait states. This improves the cache hit ratio. For example, increasing the hit ratio from 80 percent to 90 percent lets you access 10 memory locations in 20 cycles. This reduces the average number of wait states per memory access to one wait state—a substantial improvement.

Another way to improve performance is to build a *two-level* (L2) caching system. Many Intel CPUs work in this fashion. The first level is the on-chip 8,192-byte cache. The next level, between the on-chip cache and main memory, is a secondary cache (see Figure 6-21). On newer processors, the first- and second-level caches generally appear in the same packaging as the CPU. This allows the CPU designers to build a higher-performance CPU/memory interface, allowing the CPU to move data between caches and the CPU (as well as main memory) much more rapidly.

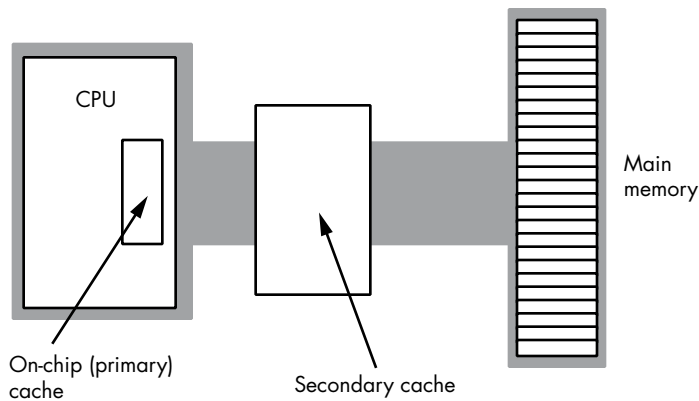


Figure 6-21: A two-level caching system

A typical on-CPU secondary cache contains anywhere from 32,768 bytes to over 2MB of memory.

Secondary cache generally does not operate at zero wait states. The circuitry to support that much fast memory would be *very* expensive, so most system designers use slower memory, which requires one or two wait states. This is still much faster than main memory. Combined with the existing on-chip L1 cache, you can get better performance from the system with a L2 caching system.

Today, many CPUs incorporate a *three-level (L3) cache*. Though the performance improvement afforded by an L3 cache is nowhere near what you get with an L1 or L2 cache subsystem, L3 cache subsystems can be quite large (usually several megabytes⁵) and work well for large systems with gigabytes of main memory. For programs that manipulate considerable data yet exhibit locality of reference, an L3 caching subsystem can be very effective.

6.5 CPU Memory Access

Most CPUs have two or three different ways to access memory. The most common *memory addressing modes* modern CPUs support are *direct*, *indirect*, and *indexed*. A few CPUs (like the 80x86) support additional addressing modes like *scaled-index*, while some RISC CPUs support only indirect access to memory. Having additional memory addressing modes makes memory access more flexible. Sometimes a particular addressing mode will allow you to access data in a complex data structure with a single instruction, where otherwise two or more instructions would be required.

RISC processors can often take three to five instructions to do what a single 80x86 instruction does. However, this does not mean that an 80x86 program will run three to five times faster. Don't forget that access to memory is very slow, usually requiring wait states. Whereas the 80x86 frequently accesses memory, RISC processors rarely do. Therefore, that RISC

5. Intel i7 CPUs in 2019, for example, support 8MB on-chip L3 caches.

processor can probably execute the first four instructions, which do not access memory at all, while the single 80x86 instruction, which does access memory, is spinning on some wait states. In the fifth instruction the RISC CPU might access memory and incur wait states of its own. If both processors execute an average of one instruction per clock cycle and have to insert 30 wait states for a main memory access, we're talking about 31 clock cycles (80x86) versus 35 clock cycles (RISC), only about a 12 percent difference.

Choosing an appropriate addressing mode often enables an application to compute the same result with fewer instructions and with fewer memory accesses, thus improving performance. Therefore, if you want to write fast and compact code, it's important to understand how an application can use the different addressing modes a CPU provides.

6.5.1 The Direct Memory Addressing Mode

The direct addressing mode encodes a variable's memory address as part of the actual machine instruction that accesses the variable. On the 80x86, direct addresses are 32-bit values appended to the instruction's encoding. Generally, a program uses the direct addressing mode to access global static variables. Here's an example in HLA assembly language:

```
static
    i:dword;

    . . .
    mov( eax, i ); // Store EAX's value into the i variable.
```

When you're accessing variables whose memory address is known prior to the program's execution, the direct addressing mode is ideal. With a single instruction, you can reference the memory location associated with the variable. On those CPUs that don't support a direct addressing mode, you may need an extra instruction (or more) to load a register with the variable's memory address prior to accessing that variable.

6.5.2 The Indirect Addressing Mode

The indirect addressing mode typically uses a register to hold a memory address (there are a few CPUs that use memory locations to hold the indirect address, but this form of indirect addressing is rare in modern CPUs).

There are a couple of advantages of the indirect addressing mode over the direct addressing mode. First, you can modify the value of an indirect address (the value being held in a register) at runtime. Second, encoding which register specifies the indirect address takes far fewer bits than encoding a 32-bit (or 64-bit) direct address, so the instructions are smaller. One disadvantage is that it may take one or more instructions to load a register with an address before you can access that address.

The following HLA sequence uses an 80x86 indirect addressing mode (brackets around the register name denote the use of indirect addressing):

```
static
    byteArray: byte[16];
```

```

    . . .
lea( ebx, byteArray ); // Loads EBX register with the address
                        // of byteArray.
mov( [ebx], al );      // Loads byteArray[0] into AL.
inc( ebx );            // Point EBX at the next byte in memory
                        // (byteArray[1]).
mov( [ebx], ah );      // Loads byteArray[1] into AH.

```

The indirect addressing mode is useful for many operations, such as accessing objects referenced by a pointer variable.

6.5.3 The Indexed Addressing Mode

The indexed addressing mode combines the direct and indirect addressing modes. Specifically, the machine instructions using this addressing mode encode both an offset (direct address) and a register in the bits that make up the instruction. At runtime, the CPU computes the sum of these two address components to create an *effective address*. This addressing mode is great for accessing array elements and for indirect access to objects like structures and records. Though the instruction encoding is usually larger than for the indirect addressing mode, the indexed addressing mode has the advantage that you can specify an address directly within an instruction without having to use a separate instruction to load the address into a register.

Here's a typical example of an HLA sequence that uses an 80x86 indexed addressing mode:

```

static
    byteArray: byte[16];
    . . .
mov( 0, ebx );          // Initialize an index into the array.
while( ebx < 16 ) do

    mov( 0, byteArray[ebx] ); // Zeros out byteArray[ebx].
    inc( ebx );              // EBX := EBX +1, move on to the
                            // next array element.

endwhile;

```

The `byteArray[ebx]` instruction in this short program demonstrates the indexed addressing mode. The effective address is the address of the `byteArray` variable plus the current value in the `EBX` register.

To avoid wasting space encoding a 32-bit or 64-bit address into every instruction that uses an indexed addressing mode, many CPUs provide a shorter form that encodes an 8-bit or 16-bit offset as part of the instruction. When using this smaller form, the register provides the base address of the object in memory, and the offset provides a fixed displacement into that data structure in memory. This is useful, for example, for accessing fields of a record or structure in memory via a pointer to that structure. The earlier

HLA example encodes the address of `byteArray` using a 4-byte address. Compare that with the following use of the indexed addressing mode:

```
lea( ebx, byteArray ); // Loads the address of byteArray into EBX.  
    .  
    .  
    .  
mov( al, [ebx+2] );    // Stores al into byteArray[2]
```

This last instruction encodes the displacement value using a single byte (rather than 4 bytes); hence, the instruction is shorter and more efficient.

6.5.4 The Scaled-Index Addressing Modes

The scaled-index addressing mode, available on several CPUs, provides two facilities above and beyond the indexed addressing mode:

- The ability to use two registers (plus an offset) to compute the effective address
- The ability to multiply one of those two registers' values by a constant (typically 1, 2, 4, or 8) prior to computing the effective address.

This addressing mode is especially useful for accessing elements of arrays whose element sizes match one of the scaling constants (see the discussion of arrays in Chapter 7 for the reasons).

The 80x86 provides a scaled-index addressing mode that takes one of several forms, as shown in the following HLA statements:

```
mov( [ebx+ecx*1], al );           // EBX is base address, ecx is index.  
mov( wordArray[ecx*2], ax );     // wordArray is base address, ecx is index.  
mov( dwordArray[ebx+ecx*4], eax ); // Effective address is combination  
                                   // of offset(dwordArray)+ebx+(ecx*4).
```

6.6 For More Information

Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Elsevier, 2012.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

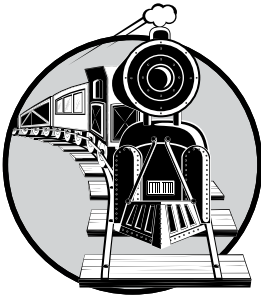
Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. Waltham, MA: Elsevier, 2014.

NOTE

Chapter 11 in this book provides additional information about cache memory and memory architecture.

7

COMPOSITE DATA TYPES AND MEMORY OBJECTS



Composite data types are composed of other, more primitive, types. Examples include pointers, arrays, records or structures, tuples, and unions. Many high-level languages (HLLs) provide syntactical abstractions for these composite data types that make them easy to declare and use, while hiding their underlying complexities.

Though the costs of using these composite data types are not terrible, a programmer who doesn't understand them can easily introduce inefficiencies into an application. This chapter provides an overview of those costs to better enable you to write great code.

7.1 Pointer Types

A *pointer* is a variable whose value refers to some other object. High-level languages like Pascal and C/C++ hide the simplicity of pointers behind a wall of abstraction. This added complexity can be intimidating if you don't

understand what’s going on behind the scenes. However, a little knowledge will go a long way toward easing your mind.

Let’s start with something simple: an array. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

M is an array with 1,024 integers in it, indexed from M[0] to M[1023]. Each array element can hold an integer value that is independent of the others. In other words, this array gives you 1,024 different integer variables, each of which you access via array index rather than by name.

The statement M[0]:=100 stores the value 100 into the first element of the array M. Now consider the following two statements:

```
i := 0; (* assume i is an integer variable *)  
M [i] := 100;
```

These two statements do the same thing as M[0]:=100;. You can use any integer expression in the range 0 through 1023 as an index of this array. The following statements *still* perform the same operation as the earlier statement:

```
i := 5;      (* assume all variables are integers*)  
j := 10;  
k := 50;  
m [i * j - k] := 100;
```

But how about the following?

```
M [1] := 0;  
M [ M [1] ] := 100;
```

Whoa! Now that takes a few moments to digest. However, if you take it slowly, you’ll realize that these two instructions perform the same operation as before. The first statement stores 0 into array element M[1]. The second statement fetches the value of M[1], which is 0, and uses that value to determine where it stores the value 100.

If you’re willing to accept this example as reasonable—perhaps bizarre, but usable nonetheless—then you’ll have no problems with pointers, because M[1] is a pointer! Well, not really, but if you were to change M to “memory” and treat each element of this array as a separate memory location, then this meets the definition of a pointer—that is, a memory variable whose value is the address of some other memory object.

7.1.1 Pointer Implementation

Although most languages implement pointers using memory addresses, a pointer is actually an abstraction of a memory address. Therefore, a language could define a pointer using any mechanism that maps the value

of the pointer to the address of some object in memory. Some implementations of Pascal, for example, use offsets from some fixed memory address as pointer values. Some languages (including dynamic languages like LISP) implement pointers by using *double indirection*; that is, the pointer object contains the address of some memory variable whose value is the address of the object to be accessed. This approach may seem convoluted, but it offers certain advantages in a complex memory management system. However, for simplicity's sake, this chapter will assume that, as defined earlier, a pointer is a variable whose value is the address of some other object in memory.

As you've seen in examples from previous chapters, you can indirectly access an object using a pointer with two 32-bit 80x86 machine instructions (or with a similar sequence on other CPUs), as follows:

```
mov( PointerVariable, ebx ); // Load the pointer variable into a register.
mov( [ebx], eax );          // Use register indirect mode to access data.
```

Access to data via double indirection is less efficient than the straight pointer implementation because it takes an extra machine instruction to fetch the data from memory. This isn't obvious in an HLL like C/C++ or Pascal, where you'd use double indirection as follows:

```
i = **cDb1Ptr;           // C/C++
i := ^^pDb1Ptr;          (* Pascal *)
```

This looks very similar to single indirection. In assembly language, however, you'll see the extra work involved:

```
mov( hDb1Ptr, ebx ); // Get the pointer to a pointer.
mov( [ebx], ebx );   // Get the pointer to the value.
mov( [ebx], eax );    // Get the value.
```

Contrast this with the two earlier assembly instructions needed to access an object using single indirection. Because double indirection requires 50 percent more code than single indirection, many languages implement pointers using single indirection.

7.1.2 Pointers and Dynamic Memory Allocation

Pointers typically reference anonymous variables that you allocate on the *heap* (a region in memory reserved for dynamic storage allocation) using memory allocation/deallocation functions like `malloc()/free()` in C, `new()/dispose()` in Pascal, and `new()/delete()` in C++ (note, however, that C++11 and later prefer `std::unique_ptr` and `std::shared_ptr` for memory allocation, with automatic memory deallocation). Java, Swift, C++11 (and later) and other more modern languages only provide a function equivalent to `new()`. These languages handle deallocation automatically via garbage collection.

Objects you allocate on the heap are known as *anonymous variables* because you refer to them by their address rather than by a name. And because the allocation functions return the address of an object on the heap, you

typically store the function's return result into a pointer variable. While the pointer variable may have a name, that name applies to the pointer's data (an address), not the object referenced by this address.

7.1.3 Pointer Operations and Pointer Arithmetic

Most languages that provide the pointer data type let you assign addresses to pointer variables, compare pointer values for equality or inequality, and indirectly reference an object via a pointer. Some languages allow additional operations, as you'll see in this section.

Many languages enable you to do limited arithmetic with pointers. At the very least, these languages provide the ability to add an integer constant to, or subtract one from, a pointer. To understand the purpose of these two arithmetic operations, note the syntax of the `malloc()` function in the C standard library:

```
ptrVar = malloc( bytes_to_allocate );
```

The parameter you pass `malloc()` specifies the number of bytes of storage to allocate. A good C programmer generally supplies an expression like `sizeof(int)` as this parameter. The `sizeof()` function returns the number of bytes needed by its single parameter. Therefore, `sizeof(int)` tells `malloc()` to allocate at least enough storage for an `int` variable. Now consider the following call to `malloc()`:

```
ptrVar = malloc( sizeof( int ) * 8 );
```

If the size of an integer is 4 bytes, this call to `malloc()` will allocate storage for 32 bytes, at consecutive addresses in memory (see Figure 7-1).

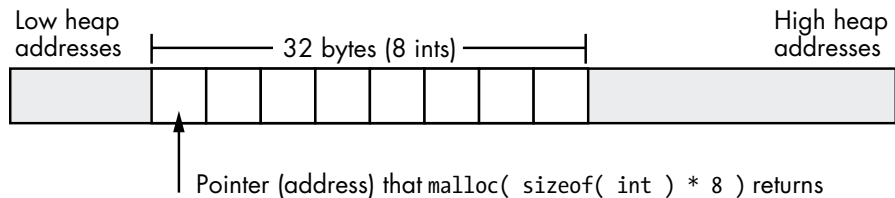


Figure 7-1: Memory allocation with `malloc(sizeof(int) * 8)`

The pointer that `malloc()` returns contains the address of the first integer in this set, so the C program can directly access only the very first of these eight integers. To access the individual addresses of the other seven integers, you need to add an integer offset to that *base* address. On machines that support byte-addressable memory (such as the 80x86), the address of each successive integer in memory is the address of the previous integer plus the integer's size. For example, if a call to the C standard library `malloc()` routine returns the memory address `$0300_1000`, then the eight integers that `malloc()` allocates will reside at the memory addresses shown in Table 7-1.

Table 7-1: Integer Addresses Allocated for Base Address \$0300_1000

Integer	Memory addresses
0	\$0300_1000..\$0300_1003
1	\$0300_1004..\$0300_1007
2	\$0300_1008..\$0300_100b
3	\$0300_100c..\$0300_100f
4	\$0300_1010..\$0300_1013
5	\$0300_1014..\$0300_1017
6	\$0300_1018..\$0300_101b
7	\$0300_101c..\$0300_101f

7.1.3.1 Adding an Integer to a Pointer

Because these integers described in the preceding section are exactly 4 bytes apart, we add 4 to the address of the first integer to obtain the address of the second integer; add 4 to the address of the second integer to get the address of the third integer; and so on. In assembly language, we could access these eight integers using the following code:

```
malloc( @size( int32 ) * 8 ); // Returns storage for eight int32 objects.
                                // EAX points at this storage.

mov( 0, ecx );
mov( ecx, [eax] );           // Zero out the 32 bytes (4 bytes
mov( ecx, [eax+4] );         // at a time).
mov( ecx, [eax+8] );
mov( ecx, [eax+12] );
mov( ecx, [eax+16] );
mov( ecx, [eax+20] );
mov( ecx, [eax+24] );
mov( ecx, [eax+28] );
```

Notice the use of the 80x86 indexed addressing mode to access the eight integers that `malloc()` allocates. The EAX register maintains the base (first) address of the eight integers that this code allocates, and the constant in the addressing mode of the `mov()` instructions selects the offset of the specific integer from this base address.

Most CPUs use byte addresses for memory objects. Therefore, when a program allocates multiple copies of some n -byte object in memory, the objects won't begin at consecutive memory addresses; instead, they'll appear in memory at addresses that are n bytes apart. Some machines, however, don't allow a program to access memory at an arbitrary address in memory; rather, they require it to access data on address boundaries that are a multiple of a word, a double word, or even a quad word. Any attempt to access memory on some other boundary will raise an exception and potentially halt the application. If an HLL supports pointer arithmetic, it must account for this fact and provide a generic pointer arithmetic scheme that's portable across many different CPU architectures. The most common

solution that HLLs use when adding an integer offset to a pointer is to multiply that offset by the size of the object that the pointer references. That is, if you've got a pointer p to a 16-byte object in memory, then $p + 1$ points 16 bytes beyond the address where p points. Likewise, $p + 2$ points 32 bytes beyond that address. As long as the size of the data object is a multiple of the required alignment size (which the compiler can enforce by adding padding bytes, if necessary), this scheme avoids problems on those architectures that require aligned data access.

Note that the addition operator only makes sense between a pointer and an integer value. For example, in C/C++ you can indirectly access objects in memory using an expression like $*(p + i)$ (where p is a pointer to an object and i is an integer value). It doesn't make sense to add two pointers together, or to add other data types to a pointer. For example, adding a floating-point value to a pointer isn't logical. (What would it mean to reference the data at some base address plus 1.5612?) Integers—signed and unsigned—are the only reasonable values to add to a pointer.

On the other hand, not only can you add an integer to a pointer, but you can also add a pointer to an integer and the result is still a pointer (both $p + i$ and $i + p$ are legal). This is because addition is *commutative*—the order of the operands does not affect the result.

7.1.3.2 Subtracting an Integer from a Pointer

Subtracting an integer from a pointer references a memory location immediately before the address held in the pointer. However, subtraction is not commutative, and subtracting a pointer from an integer is not a legal operation ($p - i$ is legal, but $i - p$ is not).

In C/C++ $*(p - i)$ accesses the i th object immediately before the object at which p points. In 80x86 assembly language, as in assembly on many processors, you can also specify a negative constant offset when using an indexed addressing mode. For example:

```
mov( [ebx-4], eax );
```

Keep in mind that 80x86 assembly language uses byte offsets, not object offsets (as C/C++ does). Therefore, this statement loads into EAX the double word in memory immediately preceding the memory address in EBX.

7.1.3.3 Subtracting a Pointer from a Pointer

In contrast to addition, it makes sense to subtract the value of one pointer variable from another. Consider the following C/C++ code, which proceeds through a string of characters looking for the first `e` character that follows the first `a` that it finds:

```
int distance;  
char *aPtr;  
char *ePtr;  
. . .
```

```

aPtr = someString;    // Get ptr to start of string in aPtr.

// While we're not at the end of the string and the current
// char isn't 'a':

while( *aPtr != '\0' && *aPtr != 'a' )
{
    aPtr = aPtr + 1; // Move on to the next character pointed
                    // at by aPtr.
}

// While we're not at the end of the string and the current
// character isn't 'e':
ePtr = aPtr;          // Start at the 'a' char (or end of string
                    // if no 'a').
while( *ePtr != '\0' && *ePtr != 'a' )
{
    ePtr = ePtr + 1; // Move on to the next character pointed at by aPtr.
}

// Now compute the number of characters between the 'a' and the 'e'
// (counting the 'a' but not counting the 'e'):

distance = (ePtr - aPtr);

```

Subtracting one pointer from the other produces the number of data objects that exist between them (in this case, `ePtr` and `aPtr` point at characters, so the subtraction result produces the number of characters, or bytes, between the two pointers).

The subtraction of two pointer values makes sense only if they both reference the same data structure (for example, pointing at characters within the same string, as in this C/C++ example) in memory. Although C/C++ (and certainly assembly language) will allow you to subtract two pointers that point at completely different objects in memory, the result will probably have very little meaning.

For pointer subtraction in C/C++, the base types of the two pointers must be identical (that is, the two pointers must contain the addresses of two objects whose types are identical). This restriction exists because pointer subtraction in C/C++ produces the number of objects, not the number of bytes, between the two pointers. It wouldn't make any sense to compute the number of objects between a byte in memory and a double word in memory; would you be counting the number of bytes or the number of double words? In assembly language you can get away with this (and the result is always the number of bytes between the two pointers), but it still doesn't make much sense semantically.

The subtraction of two pointers could return a negative number if the left pointer operand is at a lower memory address than the right pointer operand. Depending on your language and its implementation, you may need to take the absolute value of the result if you're interested only in the distance between the two pointers and you don't care which pointer contains the greater address.

7.1.3.4 Comparing Pointers

Almost every language that supports pointers will let you compare two pointers to see whether or not they are equal. Comparing two pointers will tell you whether they reference the same object in memory. Some languages (such as assembly and C/C++) will also let you compare two pointers to see if one pointer is less than or greater than the other. Such a comparison only makes sense, however, if both pointers have the same base type and contain the address of some object within the same data structure (such as an array, string, or record). If you find that one pointer is less than the other, this tells you that it references an object within the data structure that appears before the object referenced by the second pointer. The converse is true for the greater-than comparison.

7.2 Arrays

After strings, arrays are probably the most common composite (or *aggregate*) data type. Abstractly, an array is an aggregate data type whose members (elements) are all of the same type. You select a member from the array by specifying its array index with an integer (or with some value whose underlying representation is an integer, such as character, enumerated, and Boolean types). In this chapter, we'll assume that the integer indices of an array are numerically contiguous (though this is not required). That is, if both x and y are valid indices of the array, and if $x < y$, then all i such that $x < i < y$ are also valid indices. We'll also assume that array elements occupy contiguous locations in memory. Therefore, an array with five elements will appear in memory as shown in Figure 7-2.

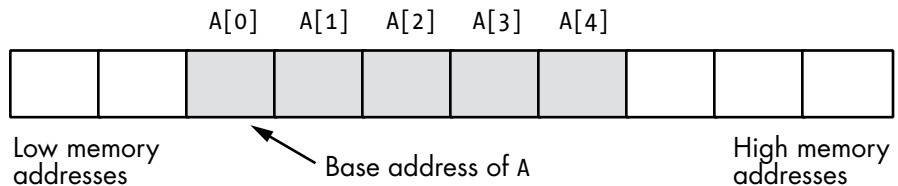


Figure 7-2: Array layout in memory

The *base address* of an array is the address of its first element and occupies the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, and so on. There is no requirement that the indices start at 0; they can start with any number as long as they're contiguous. However, we'll begin arrays at index 0 unless there's a good reason to do otherwise.

Whenever you apply the indexing operator to an array, the result is the array element specified by that index. For example, $A[i]$ chooses the i th element from array A .

7.2.1 Array Declarations

Array declarations are very similar across many HLLs. C, C++, and Java all let you declare an array by specifying the total number of elements in it. The syntax for an array declaration in these languages is as follows:

```
data_type array_name [ number_of_elements ];
```

Here are some sample C/C++ array declarations:

```
char CharArray[ 128 ];
int intArray[ 8 ];
unsigned char ByteArray[ 10 ];
int *PtrArray[ 4 ];
```

If you declare these arrays as automatic variables, then C/C++ “initializes” them with whatever bit patterns exist in memory. If, on the other hand, you declare these arrays as static objects, then C/C++ zeros out each array element. If you want to initialize an array yourself, you can use the following C/C++ syntax:

```
data_type array_name[ number_of_elements ] = {element_list};
```

Here’s a typical example:

```
int intArray[8] = {0,1,2,3,4,5,6,7};
```

Swift array declarations are a bit different from other C-based languages. Swift array declarations take one of the following two (equivalent) forms:

```
var array_name = Array<element_type>()
var array_name = [element_type]()
```

Unlike other languages, arrays in Swift are purely dynamic. You don’t normally specify the number of elements when you first create the array; instead, you add elements to the array as needed using functions like `append()` or `insert()`. If you want to predeclare an array with some number of elements, you use this special array constructor form:

```
var array_name = Array<element_type>( repeating: initial_value, count: elements)
```

In this example, *initial_value* is a value of type *element_type* and *elements* is the number of array elements to create in the array. For example, the following Swift code creates two arrays of 100 Int values, each initialized to 0:

```
var intArray = Array<Int>( repeating: 0, count: 100)
var intArray2 = [Int]( repeating: 0, count: 100)
```

You can still extend the size of this array (for example, by using the `append()` function); because Swift arrays are dynamic, their size can grow or shrink at runtime.

Swift arrays can be created with initial values, as these examples demonstrate:

```
var intArray = [1, 2, 3]
var strArray = ["str1", "str2", "str3"]
```

C# arrays are also dynamic objects; though their syntax is slightly different from Swift, the concept is the same:

```
type[ ] array_name = new type[elements];
```

Here, *type* is the data type (for example, `double` or `int`), *array_name* is the array variable name, and *elements* is the number of elements to allocate in the array.

You can also initialize C# arrays in a declaration as follows (other syntaxes are possible; this is just a simple example):

```
int[ ] intArray = {1, 2, 3};
string[ ] strArray = {"str1", "str2", "str3"};
```

The array declaration syntax in HLA (High-Level Assembly) takes the following form, which is semantically equivalent to the C/C++ declaration:

```
array_name : data_type [ number_of_elements ];
```

Here are some examples of HLA array declarations that allocate storage for uninitialized arrays (the second example assumes that you have defined the integer data type in a type section of the HLA program):

```
static

CharArray: char[128];           // Character array with elements
                                // 0..127.
IntArray: integer[8];          // Integer array with elements 0..7.
ByteArray: byte[10];           // Byte array with elements 0..9.
PtrArray: dword[4];            // Double-word array with elements 0..3.
```

You can also initialize the array elements using declarations like the following:

```
RealArray: real32[8] := [ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 ];
IntegerAry: integer[8] := [ 8, 9, 10, 11, 12, 13, 14, 15 ];
```

Both of these definitions create arrays with eight elements. The first definition initializes each 4-byte `real32` array element with one of the values in the range 0.0 through 7.0. The second declaration initializes each integer array element with one of the values in the range 8 through 15.

Pascal/Delphi uses the following syntax to declare an array:

```
array_name : array[ lower_bound..upper_bound ] of data_type;
```

As in the previous examples, *array_name* is the identifier, and *data_type* is the type of each element in this array. Unlike C/C++, Java, Swift, and HLA, in Free Pascal/Delphi you specify the upper and lower bounds of the array rather than the array's size. The following are typical array declarations in Pascal:

```
type
  ptrToChar = ^char;
var
  CharArray: array[0..127] of char;           // 128 elements
  IntArray: array[ 0..7 ] of integer;         // 8 elements
  ByteArray: array[0..9] of char;             // 10 elements
  PtrArray: array[0..3] of ptrToChar;         // 4 elements
```

Although these Pascal examples start their indices at 0, Pascal does not require it. The following Pascal array declaration is also perfectly valid:

```
var
  ProfitsByYear : array[ 1998..2039 ] of real; // 42 elements
```

The program that declares this array would use indices 1998 through 2039 when accessing elements of this array, not 0 through 41.

Many Pascal compilers provide a very useful feature to help you locate defects in your programs. Whenever you access an element of an array, these compilers automatically insert code that will verify that the array index is within the bounds specified by the declaration. This extra code will stop the program if the index is out of range. For example, if an index into *ProfitsByYear* is outside the range 1998 through 2039, the program will abort with an error.¹

Generally, array indices are integer values, though some languages allow other *ordinal types* (data types that use an underlying integer representation). For example, Pascal allows char and boolean array indices. In Pascal, it's perfectly reasonable and useful to declare an array as follows:

```
alphaCnt : array[ 'A'..'Z' ] of integer;
```

You access elements of *alphaCnt* using a character expression as the array index. For example, consider the following Pascal code, which initializes each element of *alphaCnt* to 0 (assuming *ch:char* appears in the var section):

```
for ch := 'A' to 'Z' do
  alphaCnt[ ch ] := 0;
```

1. Many Pascal compilers provide an option to turn off this bounds-checking feature once your program is fully tested; doing so improves the efficiency of the resulting program.

Assembly language and C/C++ treat most ordinal values as special instances of integer values, so they are legal array indices. Most implementations of BASIC allow a floating-point number as an array index, though BASIC always truncates the value to an integer before using it as an index.²

7.2.2 Array Representation in Memory

Abstractly, an array is a collection of variables that you access using an index. Semantically, we can define an array any way we please, as long as it maps distinct indices to distinct objects in memory and always maps the same index to the same object. In practice, however, most languages use a few common algorithms that provide efficient access to the array data.

The number of bytes of storage an array consumes is the product of the number of elements multiplied by the number of bytes per element in the array. Many languages also add a few bytes of padding at the end of the array so that the total length of the array is an even multiple of a nice value like 4 or 8 (on a 32- or 64-bit machine, a compiler may append bytes to the array in order to extend its length to some multiple of the machine's word size). However, a program must *not* depend on these extra padding bytes, because they may or may not be present. Some compilers always put them in, some never do, and still others put them in depending on the type of object that immediately follows the array in memory.

Many optimizing compilers attempt to start an array at a memory address that is an even multiple of some common size like 2, 4, or 8 bytes. Effectively, this adds padding bytes before the beginning of the array or, if you prefer to think of it this way, after the previous object in memory (see Figure 7-3).

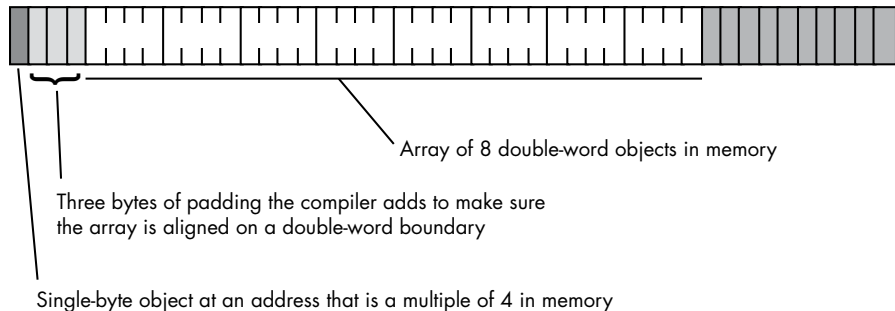


Figure 7-3: Adding padding bytes before an array

On machines that do not support byte-addressable memory, compilers that attempt to place the first element of an array on an easily accessed boundary will allocate storage for an array on whatever boundary the

2. BASIC allows you to use floating-point values as array indices because the original BASIC language did not provide support for integer expressions; it provided only real values and string values.

machine supports. If the size of each array element is less than the minimum size memory object the CPU supports, the compiler implementer has two options:

- Allocate the smallest accessible memory object for each element of the array.
- Pack multiple array elements into a single memory cell.

The first option has the advantage of being fast, but it wastes memory because each array element carries along some extra storage that it doesn't need. The second option is compact but slower, as it requires extra instructions to pack and unpack data when accessing array elements. Compilers on such machines often let you specify whether you want the data packed or unpacked so you can choose between space and speed.

If you're working on a byte-addressable machine (like the 80x86), you probably don't have to worry about this issue. However, if you're using an HLL and your code might wind up running on a different machine in the future, you should choose an array organization that is efficient on all machines.

7.2.3 Accessing Elements of an Array

If you allocate all the storage for an array in contiguous memory locations, and the first index of the array is 0, then accessing an element of a one-dimensional array is simple. You can compute the address of any given element of an array using the following formula:

$$Element_Address = Base_Address + index * Element_Size$$

Element_Size is the number of bytes that each array element occupies. Thus, if each array element is of type byte, the *Element_Size* field is 1 and the computation is very simple. If each element is a word (or another 2-byte type), then *Element_Size* is 2, and so on.

Consider the following Pascal array declaration:

```
var SixteenInts : array[ 0..15 ] of integer;
```

To access an element of the *SixteenInts* on a byte-addressable machine, assuming 4-byte integers, you'd use this calculation:

$$Element_Address = AddressOf(SixteenInts) + index * 4$$

In assembly language (where you would actually have to do this calculation manually rather than having the compiler do it for you), you'd use code like the following to access array element *SixteenInts[index]*:

```
mov( index, ebx );  
mov( SixteenInts[ ebx*4 ], eax );
```

7.2.4 Multidimensional Arrays

Most CPUs can easily handle one-dimensional arrays. Unfortunately, though, there's no magic addressing mode that lets you easily access elements of multidimensional arrays. That takes some work and several machine instructions.

Before discussing how to declare or access multidimensional arrays, let's look at how to implement them in memory. The first challenge is figuring out how to store a multidimensional object in a one-dimensional memory space.

Consider for a moment a Pascal array of the following form:

```
A:array[0..3,0..3] of char;
```

This array contains 16 bytes organized as four rows of four characters. We need to map each of the 16 bytes in this array to each of the 16 contiguous bytes in main memory. Figure 7-4 shows one way to do this.

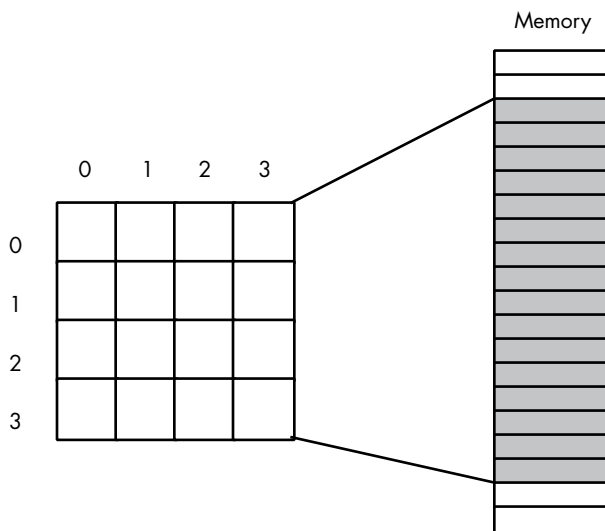


Figure 7-4: Mapping a 4x4 array to sequential memory locations

The actual mapping is not important as long as it adheres to two rules:

- No two entries in the array can occupy the same memory location(s).
- Each element in the array must always map to the same memory location.

Therefore, you need a function with two input parameters—one for a row and one for a column value—that produces an offset into a contiguous block of 16 memory locations. Any function that satisfies these two constraints will work fine. However, what you really want is a mapping function that computes efficiently at runtime and works for arrays with any

number of dimensions and any bounds on those dimensions. While there are numerous functions that fit this bill, there are two categories that most HLLs use: *row-major ordering* and *column-major ordering*.

Before I actually describe row- and column-major ordering, let's go over some terminology. The term *row index* describes a numeric index into a row; that is, if a single row were treated as a one-dimensional array, the row index would be the index into that array. *Column index* has a similar meaning; if a single column were treated as a one-dimensional array, the column index would be the index into that array. If you look back at Figure 7-4, the numbers 0, 1, 2, and 3 above each column are the *column numbers*, and those same values to the left of the rows are the *row numbers*. It's easy to get confused with this terminology because *the column number is the same value as the row index*; that is, the column number is equivalent to an index into any one of the four rows. Similarly, *a row number is the same value as a column index*. This book uses the terms *row index* and *column index*, but note that other authors may use the terms *row* and *column* to mean row number and column number.

7.2.4.1 Row-Major Ordering

Row-major ordering assigns array elements to successive memory locations by moving across a row and then down the columns. Figure 7-5 demonstrates this mapping.

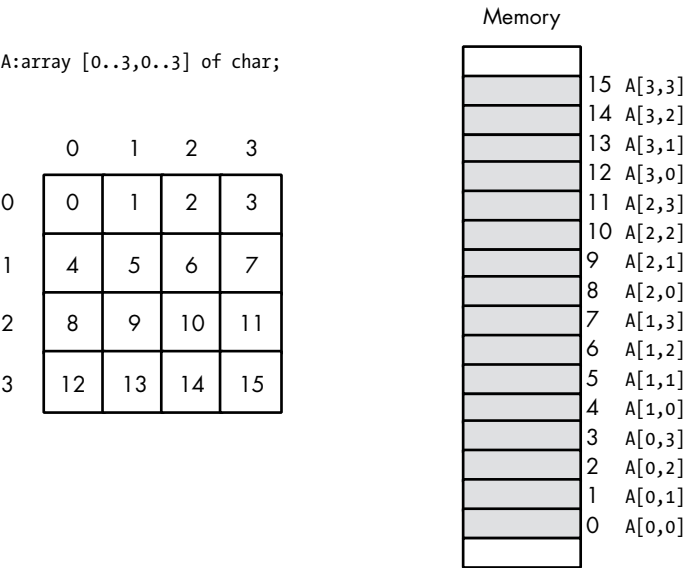


Figure 7-5: Row-major ordering

Row-major ordering is the method employed by most high-level programming languages, including Pascal, C/C++, Java, C#, Ada, and Modula-2. This organization is very easy to implement and easy to use in machine language. The conversion from a two-dimensional structure to a linear sequence is very intuitive. Figure 7-6 provides another view of the ordering of a 4×4 array.

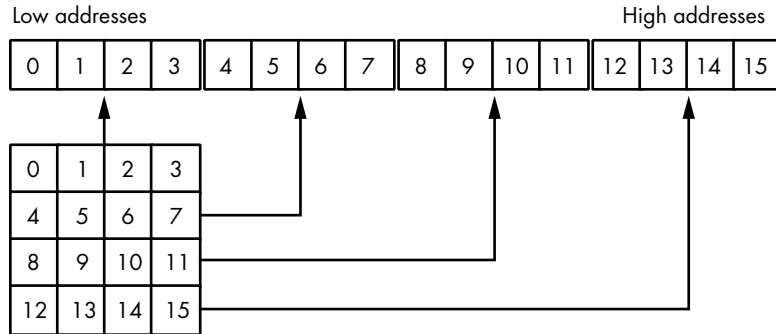


Figure 7-6: Another view of row-major ordering for a 4×4 array

The function that converts the set of multidimensional array indices into a single offset is a slight modification of the formula for computing the address of an element of a one-dimensional array. The formula to compute the offset for a 4×4 two-dimensional row-major-ordered array given an access of this form:

A[*colindex*][*rowindex*]

is as follows:

$$\text{Element_Address} = \text{Base_Address} + (\text{colindex} * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

As usual, *Base_Address* is the address of the array's first element (A[0][0] in this case) and *Element_Size* is the size of an individual element of the array, in bytes. *row_size* is the number of elements in one row of the array (4, in this case, because each row has four elements). Assuming *Element_Size* is 1, this formula computes the offsets shown in Table 7-2 from the base address.

Table 7-2: Offsets for Two-Dimensional Row-Major-Ordered Array

Column index	Row index	Offset into array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4

(continued)

Table 7-2: Offsets for Two-Dimensional Row-Major-Ordered Array (continued)

Column index	Row index	Offset into array
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

The following C/C++ code access sequential memory locations in a row-major-ordered array:

```
for( int col=0; col < 4; ++col )
{
    for( int row=0; row < 4; ++row )
    {
        A[ col ][ row ] = 0;
    }
}
```

For a three-dimensional array, the formula to compute the offset into memory is only slightly more complex. Consider the following C/C++ array declaration:

```
someType A[depth_size][col_size][row_size];
```

If you have an array access similar to `A[depth_index][col_index][row_index]`, then the computation that yields the offset into memory is:

```
Address =
Base + ((depth_index * col_size + col_index) * row_size + row_index) * Element_Size
```

Again, *Element_Size* is the size, in bytes, of a single array element.

If you've got an *n*-dimensional array declared in C/C++ as follows:

```
dataType A[bn-1][bn-2]...[b0];
```

and you wish to access the following element of this array:

```
A[an-1][an-2]...[a1][a0]
```

then you can compute the address of a particular array element using the following algorithm:

```

Address := an-1
for i := n-2 downto 0 do
    Address := Address * bi + ai
Address := Base_Address + Address * Element_Size

```

7.2.4.2 Column-Major Ordering

Column-major ordering, the other common array element address function, is used by FORTRAN and various dialects of BASIC (such as older versions of Microsoft BASIC) to index arrays. A column-major-ordered array is organized as shown in Figure 7-7.

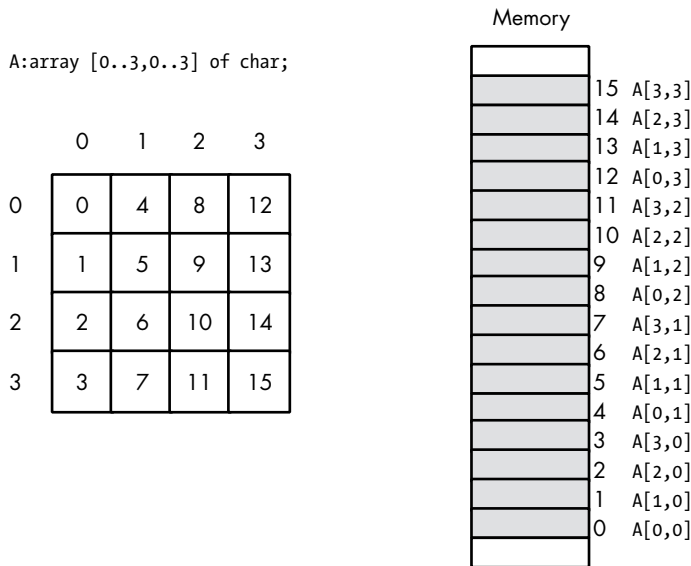


Figure 7-7: Column-major ordering

The formula for computing the address of an array element when using column-major ordering is very similar to that for row-major ordering. The difference is that you reverse the order of the index and size variables in the computation. That is, rather than working from the leftmost index to the rightmost, you operate from right to left.

For a two-dimensional column-major array, the formula is as follows:

```

Element_Address =
    Base_Address + (rowindex * col_size + colindex) * Element_Size

```

For a three-dimensional column-major array, the formula is the following:

$$\begin{aligned} \text{Element_Address} = & \\ & \text{Base_Address} + \\ & ((\text{rowindex} * \text{col_size} + \text{colindex}) * \text{depth_size} + \text{depthindex}) * \text{Element_Size} \end{aligned}$$

And so on. Other than using these new formulas, accessing elements of an array using column-major ordering is identical to accessing arrays using row-major ordering.

7.2.4.3 Declaring Multidimensional Arrays

An “ $m \times n$ ” array has $m \times n$ elements and requires $m \times n \times \text{Element_Size}$ bytes of storage. To allocate storage for an array, you must reserve this amount of memory. With one-dimensional arrays, the syntax is very similar among the different HLLs. However, their syntax starts to diverge with multidimensional arrays.

In C, C++, and Java, you use the following syntax to declare a multidimensional array:

```
data_type array_name [dim1][dim2] . . . [dimn];
```

For example, here’s a three-dimensional array declaration in C/C++:

```
int threeDInts[ 4 ][ 2 ][ 8 ];
```

This example creates an array with 64 elements organized with a depth of 4 by 2 rows by 8 columns. Assuming each int object requires 4 bytes, this array consumes 256 bytes of storage.

Pascal’s syntax supports two equivalent ways of declaring multidimensional arrays:

```
var
    threeDInts : array[0..3] of array[0..1] of array[0..7] of integer;
    threeDInts2 : array[0..3, 0..1, 0..7] of integer;
```

C# uses the following syntax to define multidimensional arrays:

```
type [,]array_name = new type [dim1,dim2] ;
type [, ,]array_name = new type [dim1,dim2,dim3] ;
type [, , ,] array_name = new type [dim1,dim2,dim3,dim4] ;
etc.
```

Semantically, there are only two major differences among different languages. The first is whether the array declaration specifies the overall size of each array dimension or the upper and lower bounds. The second is whether the starting index is 0, 1, or a user-specified value.

Swift doesn't really support multidimensional arrays in the traditional sense. It allows you to create arrays of arrays (of arrays . . .), which can provide the same functionality as multidimensional arrays, but behave in subtly different ways. See "Swift Array Implementation" on page 179 for more details.

7.2.4.4 Accessing Elements of a Multidimensional Array

It's so easy to access an element of a multidimensional array in an HLL that many programmers do so without considering the associated costs. In this section, to give you a clearer picture of those costs, we'll look at some of the assembly language sequences you'll need to access elements of a multidimensional array.

Consider again the C/C++ declaration of the `ThreeDInts` array from the previous section:

```
int ThreeDInts[ 4 ][ 2 ][ 8 ];
```

In C/C++, if you wanted to set element `[i][j][k]` of this array to `n`, you'd probably use the following statement:

```
ThreeDInts[i][j][k] = n;
```

This statement, however, hides a great deal of complexity. Recall the formula needed to access an element of a three-dimensional array:

$$\text{Element_Address} = \text{Base_Address} + ((\text{rowindex} * \text{col_size} + \text{colindex}) * \text{depth_size} + \text{depthindex}) * \text{Element_Size}$$

The `ThreeDInts` example does not avoid this calculation, it only hides it from you. The machine code that the C/C++ compiler generates is similar to the following:

```
intmul( 2, i, ebx );           // EBX = 2 * i
add( j, ebx );                 // EBX = 2 * i + j
intmul( 8, ebx );              // EBX = (2 * i + j) * 8
add( k, ebx );                 // EBX = (2 * i + j) * 8 + k
mov( n, eax );
mov( eax, ThreeDInts[ebx*4] ); // ThreeDInts[i][j][k] = n
```

Actually, `ThreeDInts` is special. The sizes of all the array dimensions are nice powers of 2. This means that the CPU can use shifts instead of multiplication instructions to multiply `EBX` by 2 and by 4 in this example. Because shifts are often faster than multiplication, a decent C/C++ compiler will generate the following code:

```
mov( i, ebx );
shl( 1, ebx );                 // EBX = 2 * i
add( j, ebx );                 // EBX = 2 * i + j
```

```

shl( 3, ebx );           // EBX = (2 * i + j) * 8
add( k, ebx );           // EBX = (2 * i + j) * 8 + k
mov( n, eax );
mov( eax, ThreeDInts[ebx*4] ); // ThreeDInts[i][j][k] = n

```

Note that a compiler can use this faster code only if an array dimension is a power of 2; this is why many programmers attempt to declare arrays with those dimensions. Of course, if you must declare extra elements in the array to achieve this goal, you may wind up wasting space (especially with higher-dimensional arrays) to achieve only a small increase in speed.

For example, if you need a 10×10 array and you’re using row-major ordering, you could create a 10×16 array to allow the use of a shift (by 4) instruction rather than a multiply (by 10) instruction. When using column-major ordering, you’d probably want to declare a 16×10 array to achieve the same effect, since row-major calculation doesn’t use the size of the first dimension when calculating an offset into an array, and column-major calculation doesn’t use the size of the second dimension when calculating an offset. In either case, however, the array would wind up having 160 elements instead of 100 elements. Only you can decide if this extra space is worth the minor improvement in speed.

7.2.4.5 Swift Array Implementation

Swift arrays are different from those found in many other languages. First of all, Swift arrays are an opaque type based on struct objects (rather than just a collection of elements in memory). Swift doesn’t guarantee that array elements appear in continuous memory locations. However, the language provides the following `ContiguousArray` type specification, which guarantees they’ll appear in contiguous memory locations (as in C/C++ and other languages):

```

var array_name = ContiguousArray<element_type>()

```

So far, so good. With contiguous arrays, the storage of the actual array data matches other languages. However, when you start declaring multidimensional arrays, the similarity ends. As noted earlier, Swift doesn’t actually have multidimensional arrays; instead, it supports *arrays of arrays*.

For most programming languages, where an array object is strictly the sequence of array elements in memory, an array of arrays and a multidimensional array are the same thing. However, Swift uses descriptor (struct-based) objects to specify an array. Like string descriptors, Swift arrays consist of a data structure containing various fields (like the current number of array elements and one or more pointers to the actual array data).

When you create an array of arrays, you’re actually creating an array of these descriptors, with each pointing at a subarray. Consider the following two (equivalent) Swift array-of-arrays declarations (a1 and a2) and sample program:

```

import Foundation

var a1 = [[Int]]()

```

```

var a2 = ContiguousArray<Array<Int>>>()
a1.append( [1,2,3] )
a1.append( [4,5,6] )
a2.append( [1,2,3] )
a2.append( [4,5,6] )

print( a1 )
print( a2 )
print( a1[0] )
print( a1[0][1] )

```

Running this program produces the following output:

```

[[1, 2, 3], [4, 5, 6]]
[[1, 2, 3], [4, 5, 6]]
[1, 2, 3]
2

```

For two-dimensional arrays you would expect this type of output. However, internally, `a1` and `a2` are one-dimensional arrays with two elements each. Those two elements are array descriptors that themselves point at arrays, each containing three elements.

It is unlikely that the six array elements associated with `a2` will appear in contiguous memory locations, even though `a2` is a `ContiguousArray` type. The two array descriptors held in `a2` may appear in contiguous memory locations, but that doesn't necessarily carry over to the six data elements at which they collectively point.

Because Swift allocates arrays dynamically, the rows in a two-dimensional array could have differing element counts. Consider the following modification to the previous Swift program:

```

import Foundation

var a2 = ContiguousArray<Array<Int>>>()
a2.append( [1,2,3] )
a2.append( [4,5] )

print( a2 )
print( a2[0] )
print( a2[0][1] )

```

Running this program produces the following output:

```

[[[1, 2, 3], [4, 5]]
[1, 2, 3]
2

```

The two rows in the `a2` array have differing sizes. This could be useful or a source of defects, depending on what you're trying to accomplish.

One way to get standard multidimensional array storage in Swift is to declare a one-dimensional `ContiguousArray` with sufficient elements for all the elements of the multidimensional array. Then use the row-major (or column-major) functionality, without the element size operand, to compute the index into the array.

7.3 Records/Structures

Another major composite data structure is the Pascal *record* or C/C++ *structure*. The Pascal terminology is probably better, as it avoids confusion with the term *data structure*, so we'll generally use *record* here.

An array is *homogeneous*, meaning that its elements are all of the same type. A record, on the other hand, is *heterogeneous*—its elements can have differing types. The purpose of a record is to let you encapsulate logically related values into a single object.

Arrays let you select a particular element via an integer index. With records, you must select an element, known as a *field*, by the field's name. Each of the field names within the record must be unique; that is, you can't use the same field name two or more times in the same record. However, all field names are local to their record, and you can reuse those names elsewhere in the program.

7.3.1 Records in Pascal/Delphi

Here's a typical record declaration for a `Student` data type in Pascal/Delphi:

```
type
    Student =
        record
            Name:    string (64);
            Major:   smallint;    // 2-byte integer in Delphi
            SSN:     string (11);
            Mid1:    smallint;
            Midt:    smallint;
            Final:   smallint;
            Homework: smallint;
            Projects: smallint;
        end;
```

Many Pascal compilers allocate all of the fields in contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name,³ the next 2 bytes for the major code, the next 12 bytes for the Social Security number, and so on.

3. Pascal strings usually require an extra byte, in addition to all the characters in the string, to encode the length.

7.3.2 Records in C/C++

Here's the same declaration in C/C++:

```
typedef
struct
{
    char Name[65]; // Room for a 64-character zero-terminated string.
    short Major;   // Typically a 2-byte integer in C/C++
    char SSN[12];  // Room for an 11-character zero-terminated string.
    short Mid1;
    short Mid2;
    short Final;
    short Homework;
    short Projects
} Student;
```

Because C++ structures are actually a specialized form of the class declaration, they behave differently from C structures and may include extra data in memory that is not present in the C variant. (This is why the memory storage for structures in C++ may be different; see “Memory Storage of Records” on page 184). There are also differences in namespaces and other minor distinctions between C and C++ structures.

As it turns out, though, you can tell C++ to compile a true C struct definition using the extern “C” block as follows:

```
extern "C"
{
    struct
    {
        char Name[65]; // Room for a 64-character zero-terminated string.
        short Major;   // Typically a 2-byte integer in C/C++
        char SSN[12];  // Room for an 11-character zero-terminated string.
        short Mid1;
        short Mid2;
        short Final;
        short Homework;
        short Projects;
    } Student;
}
```

NOTE

Java doesn't support anything corresponding to the C struct—it supports only classes (see “Classes” on page 192).

7.3.3 Records in HLA

In HLA, you can also create structure types using the `record/endrecord` declaration. For example, you would encode the record from the previous sections as follows:

```
type
    Student:
        record
            Name:    char[65];    // Room for a 64-character
                                // zero-terminated string.
            Major:   int16;
            SSN:     char[12];    // Room for an 11-character
                                // zero-terminated string.
            Mid1:    int16;
            Mid2:    int16;
            Final:   int16;
            Homework: int16;
            Projects: int16;
        endrecord;
```

As you can see, the HLA declaration is very similar to the Pascal declaration. To stay consistent with the Pascal declaration, this example uses character arrays rather than strings for the `Name` and `SSN` (Social Security number) fields. In a typical HLA record declaration, you'd probably use a string type for at least the `Name` field (keeping in mind that a string variable is a 4-byte pointer).

7.3.4 Records (Tuples) in Swift

Although Swift doesn't support the concept of a record, you can simulate one using a Swift *tuple*. While Swift does not store record (tuple) elements in memory in the same way as other programming languages (see "Memory Storage of Records" on page 184), tuples are a useful construct if you want to create a composite/aggregate data type without the overhead of a class.

A Swift tuple is simply a list of values in the following form:

```
( value1, value2, ..., valuen )
```

The types of the values within the tuple don't need to be identical.

Swift typically uses tuples to return multiple values from functions. Consider the following short Swift code fragment:

```
func returns3Ints()->(Int, Int, Int )
{
    return(1, 2, 3)
}
var (r1, r2, r3) = returns3Ints();
print( r1, r2, r3 )
```

The `returns3Ints()` function returns three values (1, 2, and 3). The following statement stores those three integer values into `r1`, `r2`, and `r3`, respectively:

```
var (r1, r2, r3) = returns3Ints();
```

You can also assign tuples to a single variable and access “fields” of the tuple using integer indices as the field names:

```
let rTuple = ( "a", "b", "c" )
print( rTuple.0, rTuple.1, rTuple.2 ) // Prints "a b c"
```

Of course, using field names like `.0` results in very hard-to-maintain code. While you could create records out of tuples, referring to the fields by an integer index is rarely suitable in real-world programs.

Fortunately, Swift allows you to assign each tuple field a label, which you can then use instead of an integer index to refer to the field. Consider the following Swift code fragment:

```
typealias record = ( field1:Int, field2:Int, field3:Float64 )

var r = record(1, 2, 3.0 )
print( r.field1, r.field2, r.field3 ) // prints "1 2 3.0"
```

Using Swift tuples this way is the syntactical equivalent of using a Pascal or HLA record (or a C structure). Keep in mind, however, that the storage of the tuple in memory might not map to the same layout as a record or structure in these other languages. Like arrays in Swift, tuples are an opaque type, without a guaranteed definition for how Swift will store them in memory.

7.3.5 Memory Storage of Records

The following Pascal example demonstrates a typical Student variable declaration:

```
var
    John: Student;
```

Given the earlier declaration for the Pascal Student data type, this allocates 81 bytes of storage laid out in memory as shown in Figure 7-8.

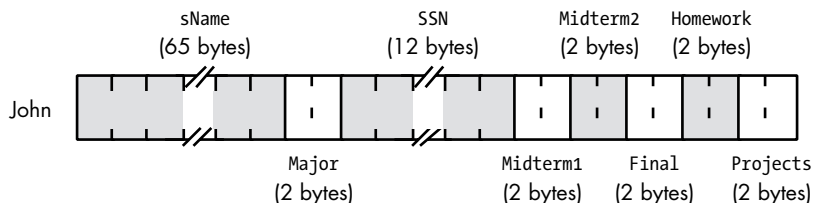


Figure 7-8: Student data structure storage in memory

If the label `John` corresponds to the *base address* of this record, then the `Name` field is at offset `John+0`, the `Major` field is at offset `John+65`, the `SSN` field is at offset `John+67`, and so on.

Most programming languages let you refer to a record field by its name rather than by its numeric offset into the record. The typical syntax for field access uses the *dot operator* to select a field from a record variable. Given the variable `John` from the previous example, here's how you could access various fields in this record:

```
John.Mid1 = 80;           // C/C++ example
John.Final := 93;         (* Pascal example *)
mov( 75, John.Projects ); // HLA example
```

Figure 7-8 suggests that all fields of a record appear in memory in the order of their declaration, and this is usually the case in practice. In theory, though, a compiler can freely place the fields anywhere in memory that it chooses. The first field usually appears at the lowest address in the record, the second field appears at the next highest address, the third field follows the second field in memory, and so on.

Figure 7-8 also suggests that compilers pack the fields into adjacent memory locations with no gaps between them. While this is true for many languages, it's certainly not the most common memory organization for a record. For performance reasons, most compilers actually align the fields of a record on appropriate memory boundaries. The exact details vary by language, compiler implementation, and CPU, but a typical compiler places fields at an offset within the record's storage area that is "natural" for that particular field's data type. On the 80x86, for example, compilers that follow the Intel ABI (application binary interface) allocate 1-byte objects at any offset within the record, words only at even offsets, and double-word or larger objects on double-word boundaries. Although not all 80x86 compilers support the Intel ABI, most do, which allows records to be shared among functions and procedures written in different languages on the 80x86. Other CPU manufacturers provide their own ABI for their processors, and programs that adhere to an ABI can share binary data at runtime with other programs that adhere to the same ABI.

In addition to aligning the fields of a record at reasonable offset boundaries, most compilers also ensure that the length of the entire record is a multiple of 2, 4, 8, or even 16 bytes. As mentioned earlier in the chapter, they accomplish this by appending padding bytes to fill out the record's size. This ensures that the record's length is an even multiple of the size of the largest scalar (noncomposite data type) object in the record or the CPU's optimal alignment size, whichever is smaller. For example, if a record has fields whose lengths are 1, 2, 4, 8, and 10 bytes, then an 80x86 compiler generally will pad the record's length so that it is an even multiple of 8. This allows you to create an array of records and be assured that each record in the array starts at a reasonable address in memory.

Although some CPUs don't allow access to objects in memory at misaligned addresses, many compilers allow you to disable the automatic alignment of fields within a record. Generally, the compiler has an option you can use to globally disable this feature. Many compilers also provide a pragma or a packed keyword that lets you turn off field alignment on a record-by-record basis. Disabling the automatic field alignment feature may save some memory by eliminating the padding bytes between the fields and at the end of the record (again, provided that field misalignment is acceptable on your CPU). However, the program may run a little bit slower when it needs to access misaligned values in memory.

One reason to use a packed record is to gain manual control over the alignment of the record's fields. For example, suppose you have a couple of functions written in two different languages, and both functions need to access some data in a record. Suppose also that the two compilers for these functions do not use the same field alignment algorithm. A record declaration like the following (in Pascal) may not be compatible with the way both functions access the record data:

```
type
  aRecord: record
    bField : byte; (* assume Pascal compiler supports a byte type *)
    wField : word; (* assume Pascal compiler supports a word type *)
    dField : dword; (* assume Pascal compiler supports a double-word type *)
  end; (* record *)
```

The problem here is that the first compiler could use the offsets 0, 2, and 4 for the bField, wField, and dField fields, respectively, while the second compiler might use offsets 0, 4, and 8.

Suppose, however, that the first compiler allows you to specify the packed keyword before the record keyword, causing the compiler to store each field immediately following the previous one. Although using the packed keyword doesn't make the records compatible with both functions, it does allow you to manually add padding fields to the record declaration, as follows:

```
type
  aRecord: packed record
    bField   :byte;
    padding0 :array[0..2] of byte; (* add padding to dword align wField *)
    wField   :word;
    padding1 :word;                (* add padding to dword align dField *)
    dField   :dword;
  end; (* record *)
```

Adding padding manually can make code maintenance a real chore. However, if incompatible compilers need to share data, it's a trick worth knowing. For the exact details on packed records, consult your language's reference manual.

7.4 Discriminant Unions

A discriminant union (or just *union*) is very similar to a record. Like records, unions have fields that you access using dot notation. In many languages, the only syntactical difference between records and unions is the use of the keyword `union` rather than `record`. Semantically, however, there's a big difference between them. In a record, each field has its own offset from the base address of the record, and the fields do not overlap. In a union, however, all fields have the same offset, 0, and all the fields of the union overlap. As a result, the size of a record is the sum of the sizes of all the fields (plus, possibly, some padding bytes), whereas a union's size is the size of its largest field (plus, possibly, some padding bytes at the end).

Because the fields of a union overlap, you might think it's of little use in a real-world program. After all, if the fields all overlap, then changing the value of one field changes the values of all the others as well. This means that union fields are *mutually exclusive*—that is, you can use only one at a time. While it's true that this makes unions less generally applicable than records, they still have many uses.

7.4.1 Unions in C/C++

Here's an example of a union declaration in C/C++:

```
typedef union
{
    unsigned int  i;
    float         r;
    unsigned char c[4];
} unionType;
```

Assuming the C/C++ compiler allocates 4 bytes for unsigned integers, the size of a `unionType` object will be 4 bytes (because all three fields are 4-byte objects).

NOTE

Unfortunately, Java doesn't support discriminant unions due to the safety issues involved. You can implement some features of discriminant unions using subclassing, but Java does not support explicitly sharing memory locations among different variables.

7.4.2 Unions in Pascal/Delphi

Pascal/Delphi use *case-variant records* to create a discriminant union. The syntax for a case-variant record is as follows:

```
type
    typeName =
        record
            <<nonvariant/union record fields go here>>
```

```

        case tag of
            const1:( field_declaration );
            const2:( field_declaration );
            .
            .
            .
            constn:( field_declaration ) (* no semicolon follows
                                           the last field *)
        end;

```

In this example, *tag* is either a type identifier (such as boolean, char, or some user-defined type) or a field declaration of the form *identifier:type*. If the *tag* item takes this latter form, then *identifier* becomes another field of the record, not a member of the *variant section* (those declarations following the case), and has the specified *type*. In addition, the Pascal compiler could generate code that raises an exception whenever the application attempts to access any of the variant fields except the one specified by the value of the *tag* field. In practice, though, almost no Pascal compilers do this check. Still, keep in mind that the Pascal language standard suggests that compilers should do it, so some compilers out there might.

Here's an example of two different case-variant record declarations in Pascal:

```

type
    noTagRecord=
        record
            someField: integer;
            case boolean of
                true:( i:integer );
                false:( b:array[0..3] of char)
            end; (* record *)

    hasTagRecord=
        record
            case which:0..2 of
                0:( i:integer );
                1:( r:real );
                2:( c:array[0..3] of char )
            end; (* record *)

```

As you can see in the *hasTagRecord* union, a Pascal case-variant record does not require any normal record fields. This is true even if you do not have a tag field.

7.4.3 Unions in Swift

Swift does not directly support the concept of a discriminant union. Unlike Java, however, Swift does provide an alternative—equivalent to Pascal’s case-variant record—that supports the safe use of unions: enumerated data types.

Consider the following Swift enumeration definition:

```
enum EnumType
{
    case a
    case b
    case c
}

let et = EnumType.b
print( et ) // prints "b" on standard output
```

So far, this is just an enumerated data type that has nothing to do with unions. However, we can attach a value (actually, a tuple of values) to each case in an enumerated data type. Consider the following Swift program, which demonstrates enum *associated values*:

```
import Foundation

enum EnumType
{
    case isInt( Int )
    case isReal( Double )
    case isString( String )
}

func printEnumType( _ et:EnumType )
{
    switch( et )
    {
        case .isInt( let i ):
            print( i )
        case .isReal( let r ):
            print( r )
        case .isString( let s ):
            print( s )
    }
}

let etI = EnumType.isInt( 5 )
let etF = EnumType.isReal( 5.0 )
let etS = EnumType.isString( "Five" )

print( etI, etF, etS )
printEnumType( etI )
printEnumType( etF )
printEnumType( etS )
```

This program produces the following output:

```
isInt(5) isReal(5.0) isString("Five")
5
5.0
Five
```

A variable of type `EnumType` takes on one of the enumeration values `isInt`, `isReal`, or `isString` (these are the three constants of type `EnumType`). In addition to whatever internal encoding Swift chooses for these three constants (probably 0, 1, and 2, though their actual values are irrelevant), Swift associates an integer value with `isInt`, a 64-bit double-precision floating-point value with `isReal`, and a string value with `isString`. The three `let` statements assign the appropriate values to `EnumType` variables; as you can see, to assign the value you include it in parentheses after the constant's name. You can then extract the value using a `switch` statement.

7.4.4 Unions in HLA

HLA supports unions as well; here's a typical union declaration:

```
type
    unionType:
        union
            i: int32;
            r: real32;
            c: char[4];
        endunion;
```

7.4.5 Memory Storage of Unions

As noted previously, the big difference between a union and a record is the fact that records allocate storage for each field at different offsets, whereas unions overlay all of the fields at the same offset in memory. For example, consider the following HLA record and union declarations:

```
type
    numericRec:
        record
            i: int32;
            u: uns32;
            r: real64;
        endrecord;
    numericUnion:
        union
            i: int32;
            u: uns32;
            r: real64;
        endunion;
```

If you declare a variable, `n`, of type `numericRec`, you access the fields as `n.i`, `n.u`, and `n.r`, exactly as though you had declared the `n` variable to be type `numericUnion`. However, the size of a `numericRec` object is 16 bytes, because the record contains two double-word fields and a quad-word (real64) field. The size of a `numericUnion` variable, though, is 8 bytes. Figure 7-9 shows the memory arrangement of the `i`, `u`, and `r` fields in both the record and union.

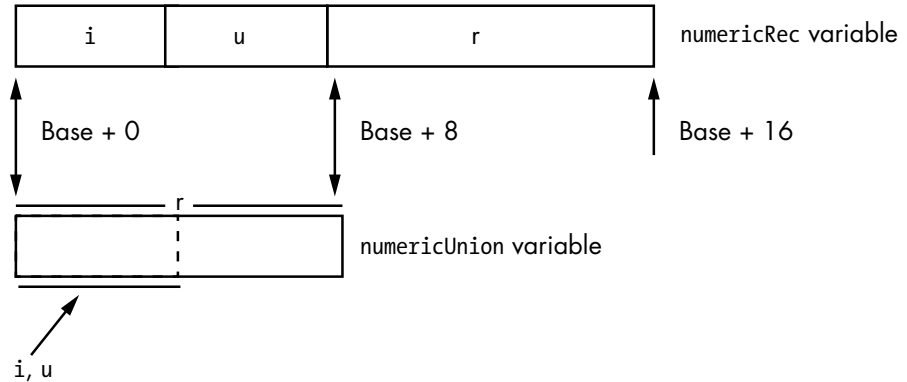


Figure 7-9: Layout of a union versus a record variable

Note that Swift `enum` types are opaque. They may not store the associated values from each enumeration case in the same memory address—and even if they currently do, there’s no guarantee they will in future versions of Swift.

7.4.6 Other Uses of Unions

In addition to conserving memory, another common reason why programmers use unions is to create aliases in their code. An *alias* is a second name for some memory object. Although aliases are often a source of confusion in a program and should be used sparingly, sometimes it’s convenient to use them. For example, in some section of your program you might need to constantly use type coercion to refer to a particular object. To avoid this, you could use a union variable with each field representing one of the different types you want to use for the object. Consider the following HLA code fragment:

```

type
    CharOrUns:
        union
            c:char;
            u:uns32;
        endunion;

static
    v:CharOrUns;

```

With a declaration like this, you can manipulate an `uns32` object by accessing `v.u`. If, at some point, you need to treat the LO byte of this `uns32` variable as a character, you can do so by simply accessing the `v.c` variable as follows:

```
mov( eax, v.u );
stdout.put( "v, as a character, is '", v.c, "'" nl );
```

Another common practice is to use unions to disassemble a larger object into its constituent bytes. Consider the following C/C++ code fragment:

```
typedef union
{
    unsigned int u;
    unsigned char bytes[4];
} asBytes;

asBytes composite;
.
.
.
composite.u = 1234567890;
printf
(
    "HO byte of composite.u is %u, LO byte is %u\n",
    composite.u[3],
    composite.u[0]
);
```

Although composing and decomposing data types this way is a useful trick to employ every now and then, keep in mind that this code isn't portable. The HO and LO bytes of a multibyte object appear at different addresses on big-endian versus little-endian machines. As a result, this code fragment works fine on little-endian machines, but fails to display the correct bytes on big-endian CPUs. Any time you use unions to decompose larger objects, you should be aware of this limitation. Still, this trick is usually much more efficient than using shift lefts, shift rights, and AND operations, so you'll see it used quite a bit.

NOTE *Swift's type safety system does not allow you to access a collection of bits as different types using discriminant unions. If you really want to convert one type to another by raw bit assignment, you can use the `Swift unsafeBitCast()` function. See the Swift standard library documentation for more details.*

7.5 Classes

At first glance, classes in a programming language like C++, Object Pascal, or Swift look like they are simple extensions to records (or structures) and should have a similar memory organization. Indeed, most programming

languages do organize class data fields in memory very similarly to records and structures. The compiler lays out the fields in sequential memory locations as it encounters them in a class declaration. However, classes have several additional features that you won't find in pure record and structures; specifically, member functions (functions declared inside a class), inheritance, and polymorphism have a big impact on how compilers implement class objects in memory.

Consider the following HLA structure and HLA class declarations:

```

type
    student: record
        sName:   char[65];
        Major:   int16;
        SSN:     char[12];
        Midterm1: int16;
        Midterm2: int16;
        Final:   int16;
        Homework: int16;
        Projects: int16;
    endrecord;

    student2: class
        var
            sName:   char[65];
            Major:   int16;
            SSN:     char[12];
            Midterm1: int16;
            Midterm2: int16;
            Final:   int16;
            Homework: int16;
            Projects: int16;

            method setName( source:string );
            method getName( dest:string );
            procedure create; // Constructor for class
        endclass;

```

As with records, HLA allocates storage for all var fields in a class sequentially. Indeed, if a class consists only of var data fields, its memory representation is nearly identical to that of a corresponding record declaration (see Figures 7-10 and 7-11).

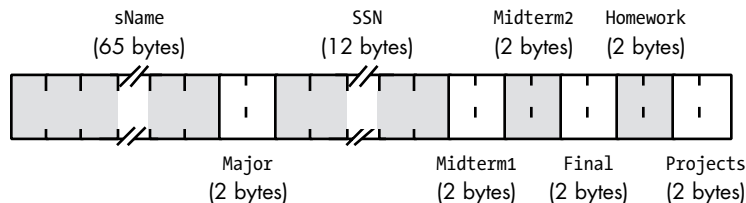


Figure 7-10: Layout of the HLA student record

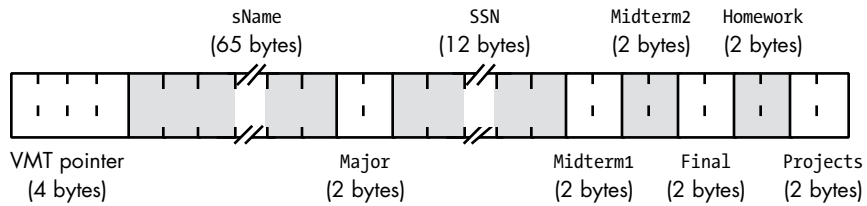


Figure 7-11: Layout of the HLA *student2* class

As you can see from these figures, the difference is the presence of the VMT pointer field at the beginning of the *student2* class data. *VMT*, which stands for *virtual method table*, is a pointer to an array of pointers to the methods (functions) associated with the class.⁴ In the *student2* example, the VMT field would point at a table containing two 32-bit pointers—one pointing at the *setName()* method and one pointing at the *getName()* method. When a program calls one of the virtual methods *setName()* or *getName()* in this class, it does not call them directly at their address in memory. Instead, it fetches the address of the VMT from the object in memory, uses that pointer to fetch the specific method address (*setName()* will likely be at the first index into the VMT and *getName()* at the second), and then use the fetched address to call the method indirectly.

7.5.1 Inheritance

Obtaining the method address from the VMT is a lot of work. Why would the compiled code do this rather than calling the method directly? The reason is because of a pair of magical features that classes and objects support: inheritance and polymorphism. Consider the following HLA class declaration:

```

type
    student3: class inherits( student2 )
        var
            extraTime: int16; // Extra time allotted for exams
            override method setName;
            override procedure create;
        endclass;

```

The *student3* class inherits all the data fields and methods from the *student2* class (as specified by the *inherits* clause in the class declaration) and then defines a new data field, *extraTime*, that allots extra time, in minutes, for the student during examinations. The *student3* declaration also defines a new method, *setName()*, that replaces the original *setName()* method in the *student2* class (it also defines an overridden *create* procedure, but we'll ignore this for now). The memory layout for a *student3* object appears in Figure 7-12.

4. Note that *create* is a class procedure, not a method. Class procedures do not appear in the VMT.

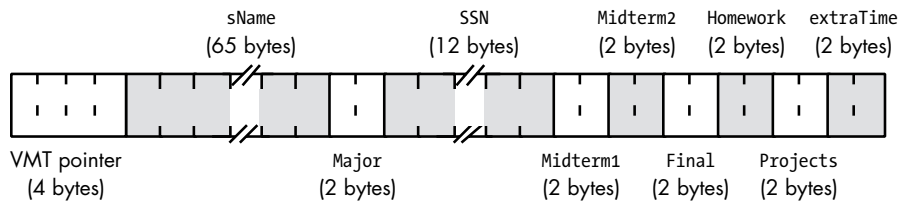


Figure 7-12: Layout of the HLA *student3* class

In memory, the difference between the *student2* and *student3* objects is the extra 2 bytes at the end of the *student3* data structure and the value held by the VMT field. For *student2* objects the VMT field points at the VMT for the *student2* class (there is only one actual *student2* VMT in memory, and all *student2* objects contain a pointer to it). If we have a pair of *student2* objects named John and Joan, their VMT fields will both contain the address of the same VMT in memory, which has the information shown in Table 7-3.

Table 7-3: VMT Entries for *student2* VMT

Offset ⁵	Entry
0 (bytes)	Pointer to the { <i>student2</i> } setName() method
4 (bytes)	Pointer to the getName() method

Now consider the case where we have a *student3* object in memory (let's name it Jenny). The memory layout for Jenny is similar to that of John and Joan (see Figures 7-11 and 7-12). However, whereas the VMT fields in John and Joan both contain the same value (a pointer to the *student2* VMT), the VMT field for the Jenny object will point at the *student3* VMT (see Table 7-4).

Table 7-4: VMT Entries for *student3* VMT

Offset	Entry
0 (bytes)	Pointer to the { <i>student3</i> } setName() method
4 (bytes)	Pointer to the getName() method

Although the *student3* VMT looks almost identical to the *student2* VMT, there is one critical difference: the first entry in Table 7-3 points at the *student2* setName() method, whereas the first entry in Table 7-4 points at the *student3* setName() method.

Adding fields inherited from a *base class* to another class must be done carefully. Remember, an important attribute of a class that inherits fields from a base class is that you can use a pointer to the base class to access

5. In assembly language, indices into the table are byte indices. Because HLA pointers are 4 bytes, each offset in the table is 4 bytes greater than the offset of the previous entry.

its fields, even if the pointer contains the address of some other class (that inherits the fields from the base class). Consider the following classes:

```
type
    tBaseClass: class
        var
            i:uns32;
            j:uns32;
            r:real32;

        method mBase;
    endclass;

    tChildClassA: class inherits( tBaseClass )
        var
            c:char;
            b:boolean;
            w:word;

        method mA;
    endclass;

    tChildClassB: class inherits( tBaseClass )
        var
            d:dword;
            c:char;
            a:byte[3];

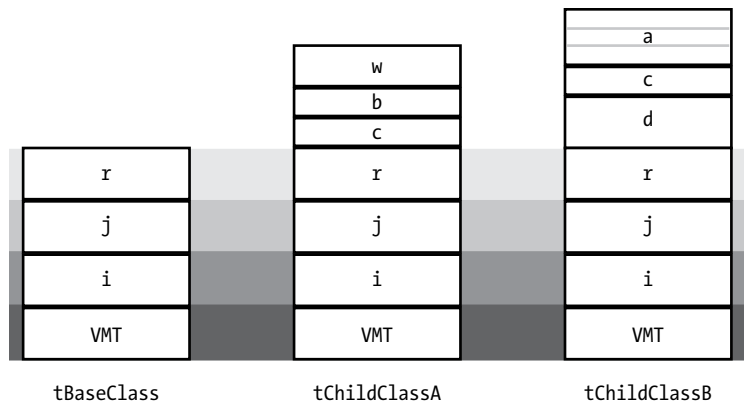
    endclass;
```

Because both `tChildClassA` and `tChildClassB` inherit the fields of `tBaseClass`, these two child classes include the `i`, `j`, and `r` fields as well as their own specific fields.

For inheritance to work properly, the `i`, `j`, and `r` fields must appear at the same offsets in all child classes as they do in `tBaseClass`. This way, an instruction of the form `mov((type tBaseClass [ebx]).i, eax);` will correctly access the `i` field even if `EBX` points at an object of type `tChildClassA` or `tChildClassB`. Figure 7-13 shows the layout of the child and base classes.

Note that the new fields in the two child classes bear no relation to one another, even if they have the same name (for example, the `c` fields in the two child classes do not lie at the same offset). Although the two child classes share the fields they inherit from their common base class, any new fields they add are unique and separate. Two fields in different classes share the same offset only by coincidence if those fields are not inherited from a common base class.

All classes (even those that aren't related to one another) place the pointer to the VMT at the same offset within the object (typically offset 0). There is a single VMT associated with each class in a program; even when classes inherit fields from some base class, their VMT (generally) differs from the base class's VMT. Figure 7-14 shows how objects of type `tBaseClass`, `tChildClassA`, and `tChildClassB` point at their specific VMTs.



Derived (child) classes locate their inherited fields at the same offsets as those fields in the base class.

Figure 7-13: Layout of base and child classes in memory

```
var
  B1: tBaseClass
  CA: tChildClassA
  CB: tChildClassB
  CB2: tChildClassB
  CA2: tChildClassA
```

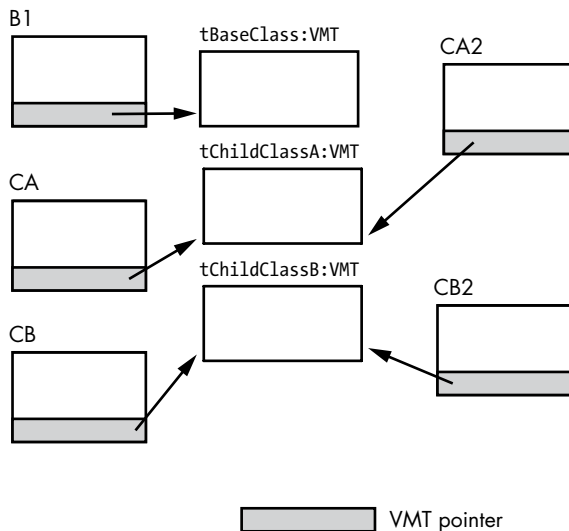


Figure 7-14: VMT references from objects

Whenever a child class inherits fields from some base class, the child class's VMT also inherits entries from the base class's VMT. For example, the VMT for the class `tBaseClass` contains only a single entry—a pointer to the method `tBaseClass.mBase()`. The VMT for the class `tChildClassA` contains

two entries: pointers to `tBaseClass.mBase()` and `tChildClassA.mA()`. Because `tChildClassB` doesn't define any new methods or iterators, its VMT contains only a single entry: a pointer to the `tBaseClass.mBase()` method. Note that `tChildClassB`'s VMT is identical to `tBaseClass`'s table. Nevertheless, HLA produces two distinct VMTs. Figure 7-15 shows this relationship.

Virtual method tables for derived (inherited) classes

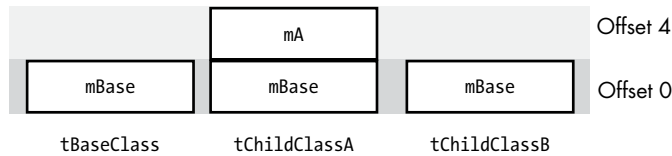


Figure 7-15: Layout of base and child classes in memory

7.5.2 Class Constructors

Before you can actually call any methods in a VMT, you have to make sure that the table is actually present in memory (holding the addresses of the methods defined in a class), and you also have to initialize the VMT pointer field in every class you create. If you're using an HLL (such as C++, C#, Java, or Swift), the compiler will automatically generate the VMTs for you when you compile the class definitions. As for initializing the VMT pointer field in the object itself, that's usually handled by the default constructor (object initialization function) for each class. All this work is hidden from an HLL programmer. That's why these class examples are using HLA—in assembly language (even a high-level assembly language), very little is hidden from you. With HLA examples, then, you get to see exactly how objects work and the cost of using them.

To begin with, HLA does not automatically create the VMTs for you. You must explicitly declare them in your code for each class you define. For the `student2` and `student3` examples, you can declare them as follows:

```
readonly
    VMT( student2 );
    VMT( student3 );
```

Technically, these don't have to appear in a `readonly` section (they could also appear in an HLA static section); however, you'll never change the VMT values, so this section is a good place to declare them.

The VMT declarations in this example define two symbols you can access in the HLA program: `student2._VMT_` and `student3._VMT_`. These symbols correspond to the address of the first entry in each VMT. Somewhere in your code (typically in the constructor procedure), you need to initialize the VMT field of the object with the address of the VMT for the associated class. The HLA convention for the class constructors appears in the following code:

```
procedure student2.create; @noframe;
begin create;
```

```

push( eax );

// ESI will contain NULL if this is called as "student2.create();"
// ESI will not be NULL if you call create from an object reference,
// such as "John.create();" (in which case ESI will point at the object,
// John in this case).

if( esi == NULL ) then

    // If a class call, allocate storage for the object
    // on the heap.

    mov( malloc( @size( student2 )), esi );

endif;
mov( &student2._VMT_, this._pVMT_ );

// If you're going to initialize other fields of the class, do that here.

pop( eax );
ret();

end create;

procedure student3.create; @noframe;
begin create;

    push( eax );
    if( esi == NULL ) then

        mov( malloc( @size( student3 )), esi );

    endif;

    // Must call the base constructor to do any class initialization
    // it requires.

    (type student2 [esi]).create(); // Must call the base class constructor.

    // Might want to initialize any student3-specific fields (such
    // as extra time) here:

    // student2.create filled in the VMT pointer with the address of the
    // student2 VMT. It really needs to point at the student3 VMT.
    // Fix that here.

    mov( &student3._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end create;

```

`student2.create()` and `student3.create()` are *class procedures* (also known as *static class methods* or *functions* in some languages). The main point to class procedures is that the code calls them directly, not indirectly (that is, through the VMT). So, if you call `John.create()` or `Joan.create()`, you're always going to call the `student2.create()` class procedure. Likewise, if you call `Jenny.create()`—or any `student3` variable's create constructor—you'll always be calling the `student3.create()` procedure.

The two statements:

```
mov( &student2._VMT_, this._pVMT_ );  
  
mov( &student3._VMT_, this._pVMT_ );
```

copy the address of the VMT (for the given class) into the VMT pointer field (`this._pVMT_`) in the objects being created.

Note the following statement in the `student3.create()` constructor:

```
(type student2 [esi]).create(); // Must call the base class constructor.
```

Upon arriving at this point, the 80x86 ESI register contains a pointer to a `student3` object. The text `(type student2 [esi])` typecasts this to a `student2` pointer. This winds up calling the parent class's constructor (in order to initialize any fields in the base class).

Finally, consider the following code:

```
var  
    John      :pointer to student2;  
    Joan      :pointer to student2;  
    Jenny      :pointer to student3;  
    .  
    .  
    .  
    student2.create(); // Equivalent to calling "new student2"  
                        // in other languages.  
    mov( esi, John ); // Save pointer to new student2  
                        // object in John  
    student2.create();  
    mov( esi, Joan );  
    student3.create();  
    mov( esi, Jenny );
```

If you look at the `_pVMT_` entries in the `John` and `Joan` objects, you'll find that they contain the address of the VMT for the `student2` class. Likewise, the `_pVMT_` field of the `Jenny` object contains the address of the VMT for the `student3` class.

7.5.3 Polymorphism

If you have an HLA `student2` variable (that is, a variable that contains a pointer to a `student2` object in memory), you can call the `setName()` method for that object using the following HLA code:

```
John.setName("John");  
Joan.setName("Joan");
```

These particular calls are examples of high-level activity taking place in HLA. The machine code that the HLA compiler emits for the first of these statements looks something like the following:

```
mov( John, esi );  
mov( (type student2 [esi])._pVMT_, edi );  
call( [edi+0] );      // Note: the offset of the setName method in the VMT is 0.
```

Here's what this code is doing:

1. The first line copies the address held in the `John` pointer into the `ESI` register. This is because most indirect accesses on the 80x86 take place in a register, not in memory variables.
2. The VMT pointer is a field in the `student2` object structure. The code needs to obtain the pointer to the `setName()` method, held in the VMT. The `_pVMT_` field of the object (which is in memory) holds the address of the VMT. Once again, we must load this into a register to access that data indirectly. The program copies the VMT pointer into the 80x86 `EDI` register.
3. The VMT (at which `EDI` now points) contains two entries. The first entry (offset 0) contains the address of the `student2.setName()` method; the second entry (offset 4) contains the address of the `student2.getName()` method. Because we want to call the `student2.setName()` method, the third instruction in this sequence calls the method at the address held in the memory location pointed at by `[edi+0]`.

As you can see, this is quite a bit more work than calling `student.setName()` directly. Why do we go through all this effort? After all, we know that `John` and `Joan` are both `student2` objects. We also know that `Jenny` is a `student3` object. So, we ought to be able to call the `student2.setName()` or `student3.setName()` methods directly. That would take only one machine instruction, which is both faster and shorter.

The reason for all this extra work is to support polymorphism. Suppose we declare a generic `student2` object:

```
var student:pointer to student2;
```

What happens when we assign the value of Jenny to student and call `student.setName()`? Well, the code sequence is identical to that for the call for John given earlier. That is, the code loads the pointer held in student into the ESI register, copies the `_pVMT_` field into the EDI register, and then jumps indirectly through the first entry of the VMT (which points at the `setName()` method). There is, however, one major difference between this example and the previous: in this case, student is pointing at a student3 object in memory. So, when the code loads the address of the VMT into the EDI register, EDI is actually pointing at the student3 VMT, not the student2 VMT (as was the case when we used the John pointer). Therefore, when the program calls the `setName()` method, it's actually calling the `student3.setName()` method, not the `student2.setName()` method. This behavior is the basis for polymorphism in modern object-oriented programming languages.

7.5.4 Abstract Methods and Abstract Base Classes

An *abstract base class* exists solely to supply a set of common fields to its derived classes. You never declare variables whose type is an abstract base class; you always use one of the derived classes. An abstract base class is a template for creating other classes, nothing more.

The only difference in syntax between a standard base class and an abstract base class is the presence of at least one abstract method declaration. An *abstract method* is a special method that does not have an actual implementation in the abstract base class. Any attempt to call that method will raise an exception. If you're wondering what possible good an abstract method could be, keep reading.

Suppose you want to create a set of classes to hold numeric values. One class could represent unsigned integers, another class could represent signed integers, a third could implement BCD values, and a fourth could support `real64` values. While you could create four separate classes that function independently of one another, doing so passes up an opportunity to make this set of classes more convenient to use. To understand why, consider the following HLA class declarations:

```
type
    uint: class
        var
            TheValue: dword;

        method put;
        << Other methods for this class >>
    endclass;

    sint: class
        var
            TheValue: dword;

        method put;
        << Other methods for this class >>
    endclass;
```

```

r64: class
    var
        TheValue: real64;

    method put;
    << Other methods for this class >>
endclass;

```

The implementation of these classes is not unreasonable. They have fields for the data, and they have a `put()` method that, presumably, writes the data to the standard output device. They probably have other methods and procedures to implement various operations on the data. There are, however, two problems with these classes, one minor and one major, both occurring because these classes do not inherit any fields from a common base class.

The minor problem is that you have to repeat the declaration of several common fields in these classes. For example, the `put()` method is declared in each class.⁶ The major problem is that this approach is not generic—that is, you can’t create a generic pointer to a “numeric” object and perform operations like addition, subtraction, and output on that value (regardless of the underlying numeric representation).

We can easily solve these two problems by turning the previous class declarations into a set of derived classes. The following code demonstrates an easy way to do this:

```

type
    numeric: class
        method put;
        << Other common methods shared by all the classes >>
    endclass;

    uint: class inherits( numeric )
        var
            TheValue: dword;

        override method put;
        << Other methods for this class >>
    endclass;

    sint: class inherits( numeric )
        var
            TheValue: dword;

        override method put;
        << Other methods for this class >>
    endclass;

    r64: class inherits( numeric )
        var

```

6. Note, by the way, that `TheValue` is not a common field, because this field has a different type in the `r64` class.

```
        TheValue: real64;

        override method put;
        << Other methods for this class >>
endclass;
```

First, by making the `put()` method inherit from `numeric`, this code encourages the derived classes to always use the name `put()`, which makes the program easier to maintain. Second, because this example uses derived classes, it's possible to create a pointer to the `numeric` type and load that pointer with the address of a `uint`, `sint`, or `r64` object. The pointer can invoke the methods found in the `numeric` class to do functions like addition, subtraction, or numeric output. Therefore, the application that uses this pointer doesn't need to know the exact data type; it deals with numeric values only in a generic fashion.

One problem with this scheme is that it's possible to declare and use variables of type `numeric`. Unfortunately, such numeric variables aren't capable of representing any type of number (notice that the data storage for the numeric fields actually appears in the derived classes). Worse, because you've declared the `put()` method in the `numeric` class, you actually have to write some code to implement that method even though you should never really call it; the actual implementation should occur only in the derived classes. While you could write a dummy method that prints an error message (or, better yet, raises an exception), you shouldn't have to resort to that. Fortunately, there's no reason to do so—if you use *abstract* methods.

The HLA *abstract* keyword, when it follows a method declaration, tells HLA that you aren't going to provide an implementation of the method for this class. Instead, all derived classes are responsible for providing a concrete implementation for the abstract method. HLA will raise an exception if you attempt to call an abstract method directly. The following code modifies the `numeric` class to convert `put()` to an abstract method:

```
type
    numeric: class
        method put; abstract;
        << Other common methods shared by all the classes >>
    endclass;
```

An abstract base class has at least one abstract method. But you don't have to make *all* methods abstract in an abstract base class; it's perfectly legal to declare some standard methods (and, of course, provide their implementation) within it.

Abstract method declarations provide a mechanism by which a base class can specify some generic methods that the derived classes must implement. If the derived classes don't provide concrete implementations of all abstract methods, that makes them abstract base classes themselves.

A little earlier, you read that you should never create variables whose type is an abstract base class. Remember, if you attempt to execute an abstract method, the program will immediately raise an exception to complain about this illegal method call.

7.6 Classes in C++

Up to this point, all the examples of classes and objects have used HLA. That made sense because the discussion concerned the low-level implementation of classes, which is something HLA illustrates well. However, you may not ever use HLA in a program you write. So now we'll look at how high-level languages implement classes and objects. As C++ was one of the earliest HLLs to support classes, we'll start with it.

Here's a variant of the `student2` class in C++:

```
class student2
{
    private:
        char    Name[65];
        short   Major;
        char    SSN[12];
        short   Midterm1;
        short   Midterm2;
        short   Final;
        short   Homework;
        short   Projects;

    protected:
        virtual void clearGrades();

    public:
        student2();
        ~student2();

        virtual void getName(char *name_p, int maxLen);
        virtual void setName(const char *name_p);
};
```

The first major difference from HLA's classes is the presence of the `private`, `protected`, and `public` keywords. C++ and other HLLs make a concerted effort to support *encapsulation* (information hiding), and these three keywords are one of the main tools C++ uses to enforce it. Scope, privacy, and encapsulation are syntactical issues that are useful for software engineering constructs, but they really don't impact the *implementation* of classes and objects in memory. Thus, since this book's focus is implementation, we'll leave further discussion of encapsulation for *WGC4* and *WGC5*.

The layout of the C++ `student2` object in memory will be very similar to the HLA variant (of course, different compilers could lay things out differently, but the basic idea of data fields and the VMT still applies).

Here's an example of inheritance in C++:

```
class student3 : public student2
{
    public:
        short extraTime;
        virtual void setName(char *name_p, int maxLen);
        student3();
        ~student3();
};
```

Structures and classes are almost identical in C++. The main difference between the two is that the default visibility at the beginning of a class is private, whereas the default visibility for struct is public. So, we could rewrite the student3 class as follows:

```
struct student3 : public student2
{
    short extraTime;
    virtual void setName(char *name_p, int maxLen);
    student3();
    ~student3();
};
```

7.6.1 Abstract Member Functions and Classes in C++

C++ has an especially weird way of declaring abstract member functions—you place “= 0;” after the function definition in the class, like so:

```
struct absClass
{
    int someDataField;
    virtual void absFunc( void ) = 0;
};
```

As with HLA, if a class contains at least one abstract function, the class is an abstract class. Note that abstract functions must also be virtual, as they must be overridden in some derived class to be useful.

7.6.2 Multiple Inheritance in C++

C++ is one of the few modern programming languages that supports *multiple inheritance*; that is, a class can inherit the data and member functions from multiple classes. Consider the following C++ code fragment:

```
class a
{
    public:
        int i;
        virtual void setI(int i) { this->i = i; }
};
```

```

class b
{
    public:
        int j;
        virtual void setJ(int j) { this->j = j; }
};

class c : public a, public b
{
    public:
        int k;
        virtual void setK(int k) { this->k = k; }
};

```

In this example, class *c* inherits all the information from classes *a* and *b*. In memory, a typical C++ compiler will create an object like that shown in Figure 7-16.

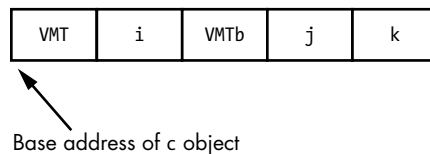


Figure 7-16: Multiple inheritance memory layout

The VMT pointer entry points at a typical VMT containing the addresses of the *setI()*, *setJ()*, and *setK()* methods (as shown in Figure 7-17). If you call the *setI()* method, the compiler will generate code that loads the *this* pointer with the address of the VMT pointer entry in the object (the base address of the *c* object in Figure 7-16). Upon entry into *setI()*, the system believes that *this* is pointing at an object of type *a*. In particular, the *this.VMT* field points at a VMT whose first (and, as far as type *a* is concerned, only) entry is the address of the *setI()* method. Likewise, at offset (*this*+4) in memory (as the VMT pointer is 4 bytes), the *setI()* method will find the *i* data value. As far as the *setI()* method is concerned, this is pointing at a class type *a* object (even though it's actually pointing at a type *c* object).

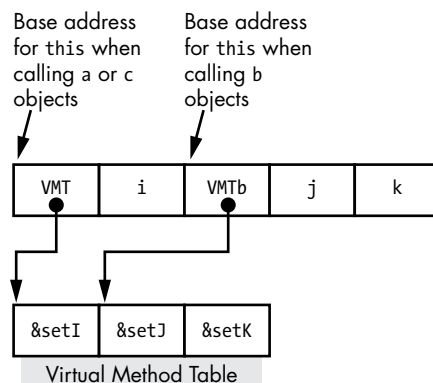


Figure 7-17: Multiple inheritance *this* values

When you call the `setK()` method, the system also passes the base address of the `c` object. Of course, `setK()` is expecting a type `c` object and this is pointing at a type `c` object, so all the offsets into the object are exactly as `setK()` expects. Note that objects of type `c` (and methods in the `c` class) will normally ignore the `VMT2` pointer field in the `c` object.

The problem occurs when the program attempts to call the `setJ()` method. Because `setJ()` belongs to class `b`, it expects this to hold the address of a VMT pointer pointing at a VMT for class `b`. It also expects to find data field `j` at offset `(this+4)`. Were we to pass the `c` object's `this` pointer to `setJ()`, accessing `(this+4)` would reference the `i` data field, not `j`. Furthermore, were a class `b` method to make a call to another method in class `b` (such as `setJ()` making a recursive call to itself), the VMT pointer would be wrong—it points at a VMT with a pointer to `setI()` at offset 0, whereas class `b` expects it to point at a VMT with a pointer to `setJ()` at offset 0. To resolve this issue, a typical C++ compiler will insert an extra VMT pointer into the `c` object immediately prior to the `j` data field. It will initialize this second VMT field to point into the `c` VMT at the location where the class `b` method pointers begin (see Figure 7-17). When calling a method in class `b`, the compiler will emit code that initializes the `this` pointer with the address of this second VMT pointer (rather than pointing at the beginning of `c`-type object in memory). Now, upon entry to a class `b` method—such as `setJ()`—this will point at a legitimate VMT pointer for class `b`, and the `j` data field will appear at the offset `(this+4)` that class `b` methods expect.

7.7 Classes in Java

Java, as a C-based language, has class definitions that are somewhat similar to C++ (though Java doesn't support multiple inheritance and has a more rational way of declaring abstract methods). Here's a sample set of Java class declarations to give you a sense of how they work:

```
public abstract class a
{
    int i;
    abstract void setI(int i);
};

public class b extends a
{
    int j;
    void setI( int i )
    {
        this.i = i;
    }

    void setJ(int j)
    {
        this.j = j;
    }
};
```

7.8 Classes in Swift

Swift is also a member of the C language tree. Like C++, Swift allows you to declare classes using the `class` or `struct` keyword. Unlike C++, Swift structures and classes are different things. A Swift structure is somewhat like a C++ class variable, whereas a Swift class is similar to a C++ pointer to an object. In Swift terminology, structures are *value* objects and classes are *reference* objects. Basically, when you create a structure object, Swift allocates sufficient memory for the entire object and binds that storage to the variable.⁷ Like Java, Swift doesn't support multiple inheritance; only single inheritance is legal. Also note that Swift doesn't support abstract member functions or classes. Here's an example of a pair of Swift classes:

```
class a
{
    var i: Int;
    init( i:Int )
    {
        self.i = i;
    }
    func setI( i :Int )
    {
        self.i = i;
    }
};

class b : a
{
    var j: Int = 0;
    override func setI( i :Int )
    {
        self.i = I;
    }
    func setJ( j:Int)
    {
        self.j = j;
    }
};
```

In Swift, all member functions are virtual by default. Also, the `init()` function is Swift's constructor. Destructors have the name `deinit()`.

7. Technically, this isn't always true. For performance reasons, Swift uses copy-on-write to improve performance; so, multiple structure objects can share the same memory location as long as you don't change the value of any field of that structure. However, once you do modify the structure, Swift makes a copy of it and changes the copy (hence the name *copy-on-write*). See the Swift documentation for more details.

7.9 Protocols and Interfaces

Java and Swift don't support multiple inheritance, because it has some logical problems. The classic example is the “diamond lattice” data structure. This occurs when two classes (say, b and c) both inherit information from the same class (say, a) and then a fourth class (say, d) inherits from both b and c. As a result, d inherits the data from a twice—once through b and once through c.

Although multiple inheritance can lead to some weird problems like this, there's no question that being able to inherit from multiple locations is often useful. Thus, the solution in languages such as Java and Swift is to allow a class to inherit methods or functions from multiple sources but to inherit data fields from only a single ancestor class. This avoids most of the problems with multiple inheritance (specifically, an ambiguous choice of inherited data fields) while allowing programmers to include methods from various sources. Java calls such extensions *interfaces*, and Swift calls them *protocols*.

Here's an example of a couple of Swift protocol declarations and a class supporting that protocol:

```
protocol someProtocol
{
    func doSomething()->Void;
    func doSomethingElse() ->Void;
}
protocol anotherProtocol
{
    func doThis()->Void;
    func doThat()->Void;
}

class supportsProtocols: someProtocol, anotherProtocol
{
    var i:Int = 0;
    func doSomething()->Void
    {
        // appropriate function body
    }
    func doSomethingElse()->Void
    {
        // appropriate function body
    }
    func doThis()->Void
    {
        // appropriate function body
    }
    func doThat()->Void
    {
        // appropriate function body
    }
}}
```

Swift protocols don't supply any functions. Instead, a class that supports a protocol promises to provide an implementation of the functions the protocol(s) specify. In the preceding example, the `supportsProtocols` class is responsible for supplying all functions required by the protocols it supports. Effectively, protocols are like abstract classes containing only abstract methods—the inheriting class must provide actual implementations for all the abstract methods.

Here's the previous example coded in Java and demonstrating its comparable mechanism, the interface:

```
class InterfaceDemo {
    interface someInterface
    {
        public void doSomething();
        public void doSomethingElse();
    }
    interface anotherInterface
    {
        public void doThis();
        public void doThat();
    }
}

class supportsInterfaces implements someInterface, anotherInterface
{
    int i;
    public void doSomething()
    {
        // appropriate function body
    }
    public void doSomethingElse()
    {
        // appropriate function body
    }
    public void doThis()
    {
        // appropriate function body
    }
    public void doThat()
    {
        // appropriate function body
    }
}

public static void main(String[] args) {
    System.out.println("InterfaceDemo");
}
}
```

Interfaces and protocols behave somewhat like base class types in Java and Swift. If you instantiate a class object and assign that instance to a variable that is an interface/protocol type, you can execute the supported member functions for that interface or protocol. Consider the following Java example:

```
someInterface some = new supportsInterfaces();

// We can call the member functions defined for someInterface:

some.doSomething();
some.doSomethingElse();

// Note that it is illegal to try and call doThis
// or doThat (or access the i data field) using
// the "some" variable.
```

Here's a comparable example in Swift:

```
import Foundation

protocol a
{
    func b()->Void;
    func c()->Void;
}

protocol d
{
    func e()->Void;
    func f()->Void;
}

class g : a, d
{
    var i:Int = 0;

    func b()->Void {print("b")}
    func c()->Void {print("c")}
    func e()->Void {print("e")}
    func f()->Void {print("f")}

    func local()->Void {print( "local to g" )}
}

var x:a = g()
x.b()
x.c()
```

You implement a protocol or interface using a pointer to a VMT that contains the addresses of the functions declared in that protocol or interface. So, the data structure for the Swift `g` class in the previous example would have three VMT pointers in it—one for protocol `a`, one for protocol `d`, and one for the class `g` (holding a pointer to the `local()` function).

When you create a variable whose type is a protocol/interface (`x` in the previous example), the variable holds the VMT pointer for that protocol. In the current example, the assignment of `g()` to the `x` variable actually just copies the VMT pointer for protocol `a` into `x`. Then, when the code executes `x.b` and `x.c`, it obtains the addresses of the actual functions from the VMT.

7.10 Generics and Templates

Although classes and objects allow software engineers to extend their systems in ways that aren't possible without object-oriented programming, objects don't provide a completely generic solution. *Generics*, first introduced by the ML programming language in 1973 and popularized by the Ada programming language, provide the key missing feature to extensibility that plain object-oriented programming was missing. Today, most modern programming languages—C++ (templates), Swift, Java, HLA (via macros), and Delphi—support some form of generic programming. In the generic programming style, you develop algorithms that operate on arbitrary data types to be defined in the future, and supply the actual data type immediately prior to using the generic type.

The classic example is a linked list. It's very easy to write a simple, singly linked list class—say, to manage a list of integers. However, after creating your list of integers, you decide you need a list of doubles. A quick copy-and-paste operation (plus changing the node type from `int` to `double`), and you've got a class that handles linked lists of double values. Oh wait, now you want a list of strings? Another cut-and-paste operation, and you've got lists of strings. Now you need a list of objects? Okay, yet another cut-and-paste. . . . You get the idea. Before too long, you've created a half-dozen different list classes and, whoops, you discover a bug in the original implementation. Now you get to go back and correct that bug in every list class you've created. Good luck with that, if you've used the list implementation in several different projects (you've just discovered why "cut and paste" programming is not considered great code).

Generics (C++ templates) come to the rescue. With a generic class definition, you specify only the algorithms (methods/member functions) that manipulate the list; you don't worry about the node type. You fill in the node type when you declare an object of the generic class type. To create integer, double, string, or object lists, you simply provide the type you want to the generic list class, and that's it. Should you discover a bug in the original (generic) implementation, all you do is fix the defect once and recompile your code; everywhere you've used the generic type, the compilation applies the correction.

Here's a C++ node and list definition:

```
template< class T >
class node {
public:
    T data;
private:
    node< T > *next;
};

template< class T >
class list {
public:
    int isEmpty();
    void append( T data );
    T remove();
    list() {
        listEnd = new node< T >();
        listEnd->next = listEnd;
    }
private:
    node< T >* listEnd;
};
```

The <T> sequence in this C++ code is a *parameterized type*. This means that you'll supply a type and the compiler will substitute that type everywhere it sees T in the template. So, in the preceding code, if you supply int as the parameter type, the C++ compiler will substitute int for every instance of T. To create a list of integers and doubles, you could use the following C++ code:

```
#include <iostream>
#include <list>

using namespace std;

int main(void) {
    list< int > integerList;
    list< double > doubleList;

    integerList.push_back( 25 );
    integerList.push_back( 0 );
    doubleList.push_back( 1.2 );
    doubleList.push_back( 3.14 );

    cout << "integerList.size() " << integerList.size() << endl;
    cout << "doubleList.size() " << doubleList.size() << endl;

    return 0;
}

doubleList.add( 3.14 );
```

The easiest way to implement generics is by using macros. When a compiler sees a declaration such as `list <int> integerList;` it expands the associated template code, substituting `int` for `T` throughout the expansion.

Because template expansion can generate a massive amount of code, modern compilers try to optimize the process wherever possible. For example, if you declare two variables like so:

```
list <int> iList1;  
list <int> iList2;
```

there's really no need to create two separate list classes, both of type `int`. Clearly, the template expansions would be identical, so any decent compiler would use the same class definition for both declarations.

Even smarter compilers would recognize that some functions, like `remove()`, don't really care about the underlying node data type. The basic removal operation is the same for all data types; as the list data type uses a pointer for the node data, there's no reason to generate different `remove()` functions for each type. With polymorphism, a single `remove()` member function would work fine. Recognizing this requires a little more sophistication on the compiler's part, but it's certainly doable.

Ultimately, however, template/generic expansion is a macro expansion process. Anything else that happens is simply an optimization by the compiler.

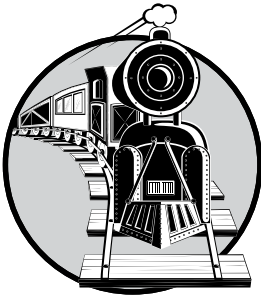
7.11 For More Information

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

Knuth, Donald. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. 3rd ed. Boston: Addison-Wesley Professional, 1997.

8

BOOLEAN LOGIC AND DIGITAL DESIGN



Boolean logic is the basis of computation in modern computer systems. You can represent any algorithm, or any electronic computer circuit, using a system of Boolean equations. To fully understand how software operates, then, you need to understand basic Boolean logic and digital design.

This material is especially important to those who want to design electronic circuits or write software that controls them. Even if you don't plan to do this, you can use your knowledge of Boolean logic to optimize your software. Many high-level languages process Boolean expressions, such as those that control an if statement or while loop. Understanding Boolean logic provides the tools you need to optimize your Boolean expressions and improve the performance of HLL code.

This chapter covers the following subjects, which will aid you when you attempt to optimize Boolean expressions:

- Boolean algebra, Boolean operators, and Boolean functions
- An introduction to Boolean postulates and theorems

- Truth tables and Boolean function optimization
- Canonical forms
- Electronic circuits and their Boolean function counterparts

Although a detailed knowledge of Boolean algebra and digital circuit design isn't necessary if you simply want to write typical programs, familiarity with these topics will help answer why CPU manufacturers implement instructions in certain ways—questions that will undoubtedly arise as we begin looking at the CPU's low-level implementation.

8.1 Boolean Algebra

Boolean algebra is a deductive mathematical system. A *binary operator* ($\circ$) accepts a pair of Boolean inputs and produces a single Boolean value. For example, the Boolean AND operator accepts two Boolean inputs and produces a single Boolean output (the logical AND of the two inputs).

8.1.1 The Boolean Operators

For our purposes, we will base Boolean algebra on the following set of values and operators:

- The two possible values in the Boolean system are 0 and 1. Often, we call these values *false* and *true*, respectively.
- The $\bullet$ symbol represents the logical AND operation. $A \bullet B$ is the operation of logically ANDing the Boolean values A and B , also known as the *product* of A and B . For single-letter variable names, this text drops the $\bullet$ symbol; therefore, AB also represents the logical AND of the variables A and B .
- The $+$ (plus sign) represents the logical OR operation. $A + B$ is the result of logically ORing the Boolean values A and B . We also call this the *sum* of A and B .
- Logical complement, logical negation, and NOT are all names for the same unary operator. This chapter will use the $'$ (prime symbol) to denote logical negation. A' denotes the logical NOT of A .

8.1.2 Boolean Postulates

Every algebraic system follows a certain set of initial assumptions, or *postulates*. You can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra employs the following postulates:

Closure A Boolean system is *closed* with respect to a particular binary operator if, for every pair of Boolean values, it produces only a Boolean result.

Commutativity A binary operator $\circ$ is *commutative* if $A \circ B = B \circ A$ for all possible Boolean values A and B .

Associativity A binary operator $\circ$ is *associative* if $(A \circ B) \circ C = A \circ (B \circ C)$ for all Boolean values A , B , and C .

Distribution Two binary operators $\circ$ and $\%$ are *distributive* if $A \circ (B \% C) = (A \circ B) \% (A \circ C)$ for all Boolean values A , B , and C .

Identity A Boolean value I is said to be the *identity element* with respect to some binary operator $\circ$ if $A \circ I = A$ for all Boolean values A .

Inverse A Boolean value I is said to be the *inverse element* with respect to some binary operator $\circ$ if $A \circ I = B$ and $B \neq A$ (that is, B is the opposite value of A in a Boolean system) for all Boolean values A and B .

When applied to the Boolean operators, the preceding postulates produce the following set of *Boolean postulates*:

P1 Boolean algebra is closed under the AND, OR, and NOT operations.

P2 The identity element of AND ($\bullet$) is 1, and the identity element of OR ($+$) is 0. There's no identity element for logical NOT ($'$).

P3 The $\bullet$ and $+$ operators are commutative.

P4 $\bullet$ and $+$ are distributive with respect to each other. That is, $A \bullet (B + C) = (A \bullet B) + (A \bullet C)$ and $A + (B \bullet C) = (A + B) \bullet (A + C)$.

P5 $\bullet$ and $+$ are both associative. That is, $(A \bullet B) \bullet C = A \bullet (B \bullet C)$ and $(A + B) + C = A + (B + C)$.

P6 For every value A there exists a value A' such that $A \bullet A' = 0$ and $A + A' = 1$. This value is the logical complement (or NOT) of A .

You can prove all other theorems in Boolean algebra using this set of Boolean postulates. This chapter won't go into the formal proofs of the following theorems, but familiarity with them will be useful:

Th1 $A + A = A$

Th2 $A \bullet A = A$

Th3 $A + 0 = A$

Th4 $A \bullet 1 = A$

Th5 $A \bullet 0 = 0$

Th6 $A + 1 = 1$

Th7 $(A + B)' = A' \bullet B'$

Th8 $(A \bullet B)' = A' + B'$

Th9 $A + A \bullet B = A$

Th10 $A \bullet (A + B) = A$

Th11 $A + A'B = A + B$

Th12 $A' \bullet (A + B') = A'B'$

Th13 $AB + AB' = A$

Th14 $(A' + B') \bullet (A' + B) = A'$

Th15 $A + A' = 1$

Th16 $A \bullet A' = 0$

NOTE

Theorems 7 and 8 are called DeMorgan's Theorems after the mathematician who discovered them.

An important principle in the Boolean algebra system is *duality*. Each pair, theorems 1 and 2, theorems 3 and 4, and so on, forms a *dual*. Any valid expression you can create using the postulates and theorems of Boolean algebra remains valid if you interchange the operators and constants appearing in the expression. Specifically, if you exchange the $\bullet$ and $+$ operators and swap the 0 and 1 values in an expression, the resulting expression will obey all the rules of Boolean algebra. *This does not mean the dual expression computes the same values*, only that both expressions are legal in the Boolean algebra system.

8.1.3 Boolean Operator Precedence

If several different Boolean operators appear within a single Boolean expression, the result of the expression depends on the *precedence* of the operators. The following Boolean operators are ordered from highest precedence to lowest:

- Parentheses
- Logical NOT
- Logical AND
- Logical OR

The logical AND and OR operators are *left associative*. This means that if two operators with the same precedence appear between three operands, you must evaluate the expressions from left to right. The logical NOT operation is *right associative*, although it would produce the same result using either left or right associativity because it is a unary operator having only a single operand.

8.2 Boolean Functions and Truth Tables

A Boolean *expression* is a sequence of 0s, 1s, and literals separated by Boolean operators. A Boolean *literal* is a primed (negated) or unprimed variable name, and all variable names are a single alphabetic character. A Boolean function is a specific Boolean expression; we generally give Boolean functions the name F with a possible subscript. For example, consider the following Boolean function:

$$F_0 = AB + C$$

This function computes the logical AND of A and B and then logically ORs this result with C . If $A = 1$, $B = 0$, and $C = 1$, then F_0 returns 1 ($1 \bullet 0 + 1 = 1$).

You can also represent a Boolean function with a *truth table*. The truth tables for the logical AND and OR functions are shown in Tables 8-1 and 8-2, respectively.

Table 8-1: AND Truth Table

AND	0	1
0	0	0
1	0	1

Table 8-2: OR Truth Table

OR	0	1
0	0	1
1	1	1

For binary operators and two input variables, this truth table format is very intuitive and convenient. However, for functions involving more than two variables, it doesn't work well.

Table 8-3 shows another way to represent truth tables. This format has several advantages—it is easier to fill in the table, it supports three or more variables, and it provides a compact representation for two or more functions.

Table 8-3: Truth Table Format for a Function of Three Variables

C	B	A	$F = ABC$	$F = AB + C$	$F = A + BC$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

Although you can create an infinite variety of Boolean functions, they are not all unique. For example, $F = A$ and $F = AA$ are two different functions. By theorem 2, however, it's easy to show that these two functions produce exactly the same result no matter what input value you supply for A . As it turns out, if you fix the number of input variables, there's a finite number of unique Boolean functions possible. For example, there are 16 unique Boolean functions with two input variables, and there are 256 possible Boolean functions with three input variables. Given n input variables, there are 2^{2^n} unique Boolean functions (2 raised to 2 raised to the n th power). With two input variables, there are 2^{2^2} or 16 different functions. With three input variables, there are 2^{2^3} or 256 possible functions. Four input variables have 2^{2^4} or 2^{16} or 65,536 unique Boolean functions.

When working with only 16 Boolean functions (two input variables), we can name each unique function (see Table 8-4).

Table 8-4: Common Names for Boolean Functions of Two Variables

Function number ¹	Function name	Description
0	Zero (clear)	Always returns 0 regardless of A and B input values.
1	Logical NOR	$(\text{NOT } (A \text{ OR } B)) = (A + B)'$
2	Inhibition (AB')	Inhibition = AB' (A AND not B). Also equivalent to $A > B$ or $B < A$.
3	NOT B	Ignores A and returns B' .
4	Inhibition (BA')	Inhibition = BA' (B AND not A). Also equivalent to $B > A$ or $A < B$.
5	NOT A	Returns A' and ignores B .
6	Exclusive-OR (XOR)	$A \oplus B$. Equivalent to $A \neq B$.
7	Logical NAND	$(\text{NOT } (A \text{ AND } B)) = (A \bullet B)'$
8	Logical AND	$A \bullet B = (A \text{ AND } B)$
9	Equivalence (exclusive-NOR)	$(A = B)$. Also known as exclusive-NOR (not exclusive-OR).
10	A	Copy A . Returns the value of A and ignores B 's value.
11	Implication, B implies A	$A + B'$. (If B then A .) Equivalent to $B \geq A$.
12	B	Copy B . Returns the value of B and ignores A 's value.
13	Implication, A implies B	$B + A'$. (If A then B .) Equivalent to $A \geq B$.
14	Logical OR	$A + B$. Returns $A \text{ OR } B$.
15	One (set)	Always returns 1 regardless of A and B input values.

8.3 Function Numbers

Beyond two input variables, there are too many functions to provide a specific name for each. Even when referring to functions with two input variables, we'll refer to the function's number rather than its name. For example, F_8 denotes the logical AND of A and B for a two-input function, and F_{14} denotes the logical OR operation. Of course, for functions with more than two input variables, the question is, "How do we determine a function's number?" For example, what is the corresponding number for the function $F = AB + C$? We compute the answer by looking at the function's truth table. If we treat the values for A , B , and C as bits in a binary number with C being the HO bit and A being the LO bit, they produce the binary strings that correspond to numbers in the range 0 through 7. Associated with each of these binary strings is the function result, either 0 or 1. If we construct a binary number by placing the function result of each

¹. See the discussion of function numbers in the next section.

combination of the A , B , and C input values into the bit position specified by the binary string of the A , B , and C bits, the resulting binary number will be the corresponding function number. If this doesn't make sense, an example will help clear it up. Consider the truth table for $F = AB + C$ (see Table 8-5).

Table 8-5: Truth Table for $F = AB + C$

C	B	A	$F = AB + C$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

The input variables C , B , and A combine to form binary number sequences in the range %000 through %111 (0 through 7). If we use these values to denote bit numbers in an 8-bit value ($CBA = \%111$ specifies bit 7, $CBA = \%110$ specifies bit 6, and so on), we can determine the function number by placing at each of these bit positions the result of $F = AB + C$, for the corresponding combination of C , B , and A values:

CBA:	7	6	5	4	3	2	1	0
$F = AB + C$:	1	1	1	1	1	0	0	0

Now, if we treat this bit string as a binary number, it produces the function number \$F8, or 248. We usually denote function numbers in decimal. This also provides insight into why there are 2^{2^n} different functions given n input variables: if you have n input variables, there are 2^n different variable value combinations, and thus 2^n bits in the function's binary number. If you have m bits, there are 2^m different possible arrangements of those bits. Therefore, for n input variables there are $m = 2^n$ possible bits and 2^m or 2^{2^n} possible functions.

8.4 Algebraic Manipulation of Boolean Expressions

You can transform one Boolean expression into an equivalent expression by applying the postulates and theorems of Boolean algebra. This is important if you want to convert a given expression to a canonical form (see the next section) or if you want to minimize the number of literals or terms in an expression. (A *literal* is a primed or unprimed variable, and a *term* is a variable or a product—logical AND—of several different literals.) Electrical circuits often consist of individual components that implement each literal

or term, so minimizing the number of literals and terms in an expression allows a circuit designer to use fewer electrical components and, therefore, to reduce the monetary cost of the system.

Unfortunately, there are no fixed rules you can apply to optimize a given expression. Much like constructing mathematical proofs, an individual's ability to easily do these transformations is usually a matter of experience. Nevertheless, a few examples show the possibilities:

$ab + ab' + a'b$	$= a(b + b') + a'b$	By P4
	$= a \bullet 1 + a'b$	By P5
	$= a + a'b$	By Th4
	$= a + b$	By Th11
$(a'b + a'b' + b')'$	$= (a'(b + b') + b')'$	By P4
	$= (a' \bullet 1 + b')'$	By P5
	$= (a' + b')$	By Th4
	$= ((ab)')'$	By Th8
	$= ab$	By definition of not
$b(a + c) + ab' + bc' + c$	$= ba + bc + ab' + bc' + c$	By P4
	$= a(b + b') + b(c + c') + c$	By P4
	$= a \bullet 1 + b \bullet 1 + c$	By P5
	$= a + b + c$	By Th4

8.5 Canonical Forms

Each Boolean function has an infinite number of equivalent logic expressions. To help eliminate confusion, logic designers generally specify a Boolean function using a *canonical*, or standardized, form. For each different Boolean function, we can choose a single canonical representation from a defined set.

There are several ways to define a set of canonical representations for all the possible Boolean functions of n variables. Within each canonical set, a single expression describes each Boolean function in the system so all of the functions in the set are unique. We'll discuss two canonical systems in this chapter—the *sum of minterms* and the *product of maxterms*—but we'll employ only the first. Using the duality principle, we can convert between these two systems.

As mentioned earlier, a term is either a single literal or a product (logical AND) of several different literals. For example, if you have two variables, A and B , there are eight possible terms: A , B , A' , B' , $A'B'$, $A'B$, AB' , and AB . For three variables, we have 26 different terms: A , B , C , A' , B' , C' , $A'B'$, $A'B$, AB' , AB , $A'C'$, $A'C$, AC' , AC , $B'C'$, $B'C$, BC' , BC , $A'B'C'$, $AB'C'$, $A'BC'$, ABC' , $A'B'BC$, $AB'BC$, $A'BC$, and ABC . As the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing

exactly n literals, where n is the number of input variables. For example, the minterms for the two variables A and B are $A'B'$, AB' , $A'B$, and AB . Likewise, the minterms for three variables A , B , and C are $A'B'C'$, $AB'C'$, $A'BC'$, ABC' , $A'B'C$, $AB'C$, $A'BC$, and ABC . In general, there are 2^n minterms for n variables. The set of possible minterms is easy to generate because they correspond to the sequence of binary numbers (see Table 8-6).

Table 8-6: Generating Minterms from Binary Numbers

Binary equivalent (CBA)	Minterm
000	$A'B'C'$
001	$AB'C'$
010	$A'BC'$
011	ABC'
100	$A'B'C$
101	$AB'C$
110	$A'BC$
111	ABC

We can derive the canonical form for *any* Boolean function using a sum (logical OR) of minterms. Given $F_{248} = AB + C$, the equivalent canonical form is $ABC + A'BC + AB'C + A'B'C + ABC'$. Algebraically, we can show that the canonical form is equivalent to $AB + C$ as follows:

$ABC + A'BC + AB'C + A'B'C + ABC'$	$= BC(A + A') + B'C(A + A') + ABC'$	By P4
	$= BC \cdot 1 + B'C \cdot 1 + ABC'$	By Th15
	$= C(B + B') + ABC'$	By P4
	$= C + ABC'$	By Th15 & Th4
	$= C + AB$	By Th11

Obviously, the canonical form is not optimal. However, it's very easy to generate the truth table for a function from the canonical form. It's also very easy to generate the sum-of-minterms canonical form equation from the truth table.

8.5.1 Sum-of-Minterms Canonical Form and Truth Tables

To build the truth table from the sum-of-minterms canonical form, follow these steps:

1. Convert minterms to binary equivalents by substituting a 1 for unprimed variables and a 0 for primed variables, like so:

$F_{248} = CBA + CBA' + CB'A + CB'A' + C'BA$
$= 111 + 110 + 101 + 100 + 011$

- Place a 1 in the function column for the appropriate minterm entries:

C	B	A	$F = AB + C$
0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

- Finally, place the number 0 in the function column for the remaining entries:

C	B	A	$F = AB + C$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Going in the other direction, to generate a logic function from a truth table, follow these steps:

- Locate all the entries in the truth table with a function result of 1. In this table, these are the last five entries. The number of table entries containing 1s determines the number of minterms in the canonical equation.
- Generate the individual minterms by substituting A , B , or C for 1s and A' , B' , or C' for 0s. In this example, the result of F_{248} is 1 when CBA equals 111, 110, 101, 100, or 011. Therefore, $F_{248} = CBA + CBA' + CB'A + CB'A' + C'AB$.
- Optionally rearrange the terms within the minterms, and rearrange the minterms within the overall function. This works because the logical OR and logical AND operations are both commutative.

This process works equally well for any number of variables, as with the truth table in Table 8-7 for the function $F_{53,504} = ABCD + A'BCD + A'B'CD + A'B'C'D$.

Table 8-7: Truth Table for $F_{53,504}$

D	C	B	A	$F = ABCD + A'BCD + A'B'CD + A'B'C'D$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Perhaps the easiest way to generate the canonical form of a Boolean function is to first generate the truth table for it and then build the canonical form from the truth table. In fact, we'll use this technique when converting between the two canonical forms.

8.5.2 Algebraically Derived Sum-of-Minterms Canonical Form

To generate the sum-of-minterms canonical form algebraically, we use the distributive law and theorem 15 ($A + A' = 1$). Consider $F_{248} = AB + C$. This function contains two terms, AB and C , but they are not minterms. We can convert the first term to a sum of minterms as follows:

AB	=	$AB \cdot 1$	By Th4
	=	$AB \cdot (C + C')$	By Th15
	=	$ABC + ABC'$	By distributive law
	=	$CBA + C'BA$	By associative law

Similarly, we can convert the second term in F_{248} to a sum of minterms as follows:

C	=	$C \cdot 1$	By Th4
	=	$C \cdot (A + A')$	By Th15
	=	$CA + CA'$	By distributive law
	=	$CA \cdot 1 + CA' \cdot 1$	By Th4

$$\begin{aligned}
&= CA \cdot (B + B') + CA' \cdot (B + B') && \text{By Th15} \\
&= CAB + CAB' + CA'B + CA'B' && \text{By distributive law} \\
&= CBA + CBA' + CB'A + CB'A' && \text{By associative law}
\end{aligned}$$

The last step (rearranging the terms) in these two conversions is optional. To obtain the final canonical form for F_{248} , we sum the results from these two conversions:

$$\begin{aligned}
F_{248} &= (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A') \\
&= CBA + CBA' + CB'A + CB'A' + C'BA
\end{aligned}$$

8.5.3 Product-of-Maxterms Canonical Form

Another canonical form is the *products of maxterms*. A maxterm is the sum (logical OR) of all input variables, primed or unprimed. For example, consider the following logic function, G , of three variables in product-of-maxterms form:

$$G = (A + B + C) \cdot (A' + B + C) \cdot (A + B' + C)$$

As with the sum-of-minterms form, there's exactly one product of maxterms for each possible logic function. For every product-of-maxterms form, there's an equivalent sum-of-minterms form. In fact, the function G in this example is equivalent to the earlier sum-of-minterms form of F_{248} :

$$F_{248} = CBA + CBA' + CB'A + CB'A' + C'BA = AB + C$$

To generate a truth table from the product of maxterms, you use the duality principle; that is, swap AND for OR and 0s for 1s (and vice versa). Therefore, to build the truth table, you'd first swap primed and nonprimed literals. In G , this would yield:

$$G = (A' + B' + C') \cdot (A + B' + C') \cdot (A' + B + C')$$

The next step is to swap the logical OR and logical AND operators, which produces the following:

$$G = A'B'C' + AB'C' + A'BC'$$

Finally, you need to swap all 0s and 1s. This means that for each of the minterms listed previously, you need to store 0s into the function column of the truth table, and then fill in the rest of the truth table's function column with 1s. This will place a 0 in rows 0, 1, and 2 in the truth table. Filling the remaining entries with 1s produces F_{248} .

You can easily convert between these two canonical forms by generating the truth table for one form and working backward to produce the other form. Consider the function of two variables, $F_7 = A + B$. The sum-of-minterms form is $F_7 = A'B + AB' + AB$. The truth table is shown in Table 8-8.

Table 8-8: OR Truth Table for Two Variables

A	B	F7
0	0	0
1	0	1
0	1	1
1	1	1

Working backward to get the product of maxterms, we first locate all entries in the truth table that have a 0 result. The entry with A and B both equal to 0 is the only entry with a 0 result. This gives us the first step of $G = A' B'$. However, we still need to invert all the variables to obtain $G = AB$. By the duality principle, we also need to swap the logical OR and logical AND operators, obtaining $G = A + B$. This is the canonical *product of maxterms* form.

8.6 Simplification of Boolean Functions

Because there's an infinite variety of Boolean functions of n variables, but a finite number of unique ones, you might wonder if there is some method that will simplify a given Boolean function to produce the optimal form—that is, the expression containing the fewest number of operators. An optimal form must exist for all logic functions, but we don't use it for the canonical form for two reasons. First, although it's easy to convert between the truth table forms and the canonical form, it's not as easy to generate the optimal form from a truth table. Second, there may be several optimal forms for a single function.

You can attempt to produce the optimal form using algebraic transformations, but there's no guarantee you'll arrive at the best result. There are two methods that will *always* reduce a given Boolean function to its optimal form: the *mapping* method and the *prime implicants* method. This book covers the mapping method.

Using the mapping method to manually optimize Boolean functions is practical only for functions of two, three, or four variables. It's doable but cumbersome for functions of five or six variables. For more than six variables, you should write a program.

The first step in the mapping method is to build a special two-dimensional truth table for the function (see Figure 8-1). *Take a careful look at these truth tables.* They do not use the same forms shown earlier in this chapter. In particular, the progression of the 2-bit values is 00, 01, 11, 10, not 00, 01, 10, 11. This is very important! If you organize the truth tables in a binary sequence, the mapping optimization method will not work properly. We'll call this a *truth map* to distinguish it from the standard truth table.²

		A				BA			
		0	1			00	01	11	10
B	0	$B'A'$	$B'A$	C	0	$C'B'A'$	$C'B'A$	$C'AB$	$C'BA'$
	1	BA'	BA		1	$CB'A'$	$CB'A$	CAB	CBA'

Two-variable truth map Three-variable truth map

		BA			
		00	01	11	10
DC	00	$D'C'B'A'$	$D'C'B'A$	$D'C'AB$	$D'C'BA'$
	01	$D'CB'A'$	$D'CB'A$	$D'CAB$	$D'CBA'$
	11	$DCB'A'$	$DCB'A$	$DCAB$	$DCBA'$
	10	$DC'B'A'$	$DC'B'A$	$DC'AB$	$DC'BA'$

Four-variable truth map

Figure 8-1: Two-, three-, and four-variable truth maps

Assuming your Boolean function is already in sum-of-minterms canonical form, insert 1s for each of the truth map cells corresponding to one of the minterms in the function. Place 0s everywhere else. For example, consider the function of three variables $F = C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA$. Figure 8-2 shows the truth map for this function.

2. These are also known as *Karnaugh maps* or *Karnaugh/Veitch diagrams*, after Maurice Karnaugh, who created them by refining Edward Veitch's Boolean optimization diagrams.

		BA			
		00	01	11	10
C	0	0	1	1	1
	1	1	1	1	1

Figure 8-2: A truth map for $F = C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA$

The next step is to draw outlines around rectangular groups of 1s. The rectangles you enclose must have sides whose lengths are powers of 2. For functions with three variables, the rectangles can have sides whose lengths are 1, 2, and 4. The set of rectangles you draw must surround all cells containing 1s in the truth map. The trick is to draw all possible rectangles unless a rectangle would be completely enclosed within another, but also draw the fewest number of rectangles. Note that the rectangles may overlap as long as one rectangle does not completely enclose the other. In the truth map in Figure 8-3, there are three such rectangles.

		BA			
		00	01	11	10
C	0	0	1	1	1
	1	1	1	1	1

Three possible rectangles whose lengths and widths are powers of 2

Figure 8-3: Surrounding rectangular groups of 1s in a truth map

Each rectangle represents a term in the simplified Boolean function. Therefore, the simplified Boolean function will contain only three terms. You build each term by eliminating any variables whose primed and unprimed forms both appear within the rectangle (because the positive and negative variants cancel each other out). The long skinny rectangle in Figure 8-3 is sitting in the row where $C = 1$ contains both A and B in primed and unprimed forms. Therefore, we can eliminate both A and B from the term. Because the rectangle sits in the $C = 1$ region, this rectangle represents the single literal C .

The light gray square in Figure 8-3 includes C , C' , B , B' , and A . Therefore, it represents the single term A . Likewise, the dark gray square in Figure 8-3 contains C , C' , A , A' , and B , so it represents the single term B .

The final, optimal, function is the sum (logical OR) of the terms represented by the three squares, or $F = A + B + C$. You do not have to consider the remaining squares containing 0s.

A truth map forms a *torus* (a doughnut shape). The right edge of the map wraps around to the left edge, and vice versa. Likewise, the top edge wraps around to the bottom edge. This introduces additional possibilities for drawing rectangles around groups of 1s in a map. Consider the Boolean function $F = C'B'A' + C'BA' + CB'A' + CBA'$. Figure 8-4 shows the truth map for this function.

		BA			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

Figure 8-4: Truth map for $F = C'B'A' + C'BA' + CB'A' + CBA'$

At first glance, you might think that the minimum number of rectangles is two, as shown in Figure 8-5.

		BA			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

Figure 8-5: First attempt at surrounding rectangles formed by 1s

However, because the truth map is a continuous object with the right side and left sides connected, we can actually form a single, square rectangle, as Figure 8-6 shows.

	BA			
	00	01	11	10
0	1	0	0	1
1	1	0	0	1

Figure 8-6: Correct rectangle for the function

Why does it matter if we have one rectangle or two in the truth map? The larger the rectangles are, the more terms they will eliminate. The fewer rectangles that we have, then, the fewer terms will appear in the final Boolean function.

The example in Figure 8-5 with two rectangles generates a function with two terms. The rectangle on the left eliminates the C variable, leaving $A'B'$ as its term. The rectangle on the right also eliminates the C variable, leaving the term BA' . Therefore, this truth map would produce the equation $F = A'B' + A'B$. We know this is not optimal (see theorem 13).

Now consider the truth map in Figure 8-6. Here we have a single rectangle, so our Boolean function will have only a single term. Because this rectangle includes both C and C' , and also B and B' , the only term left is A' . This Boolean function, therefore, reduces to $F = A'$.

There are only two types of truth maps that the mapping method cannot handle properly: a truth map that contains all 0s or a truth map that contains all 1s. These two cases correspond to the Boolean functions $F = 0$ and $F = 1$ (that is, the function number is 0 or $2^n - 1$). When you see either of these truth maps, you'll know how to optimally represent the function.

When optimizing Boolean functions using the mapping method, remember that you always want to pick the largest rectangles whose sides' lengths are powers of 2. You must do this even for overlapping rectangles (unless one rectangle encloses another). Consider the Boolean function $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$. This produces the truth map in Figure 8-7.

	BA			
	00	01	11	10
0	1	0	1	1
1	1	0	1	1

Figure 8-7: Truth map for $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$

The initial temptation is to create one of the sets of rectangles found in Figure 8-8. However, the correct mapping appears in Figure 8-9.

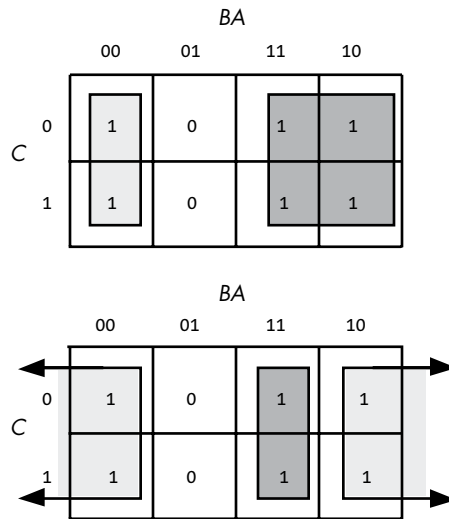


Figure 8-8: Obvious choices for rectangles

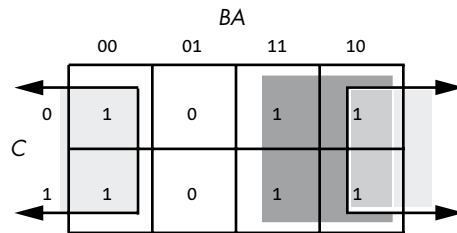


Figure 8-9: Correct set of rectangles for $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$

All three mappings will produce a Boolean function with two terms. However, the first two will produce the expressions $F = B + A'B'$ and $F = AB + A'$. The third form produces $F = B + A'$. This last form is the optimized one (see theorems 11 and 12).

Truth maps you create for functions of four variables are even trickier; there are many places rectangles can hide from you along the edges, as you can see in Figure 8-10. This list of patterns doesn't even begin to cover all of them! For example, the diagrams in Figure 8-10 show none of the 1x2 rectangles.

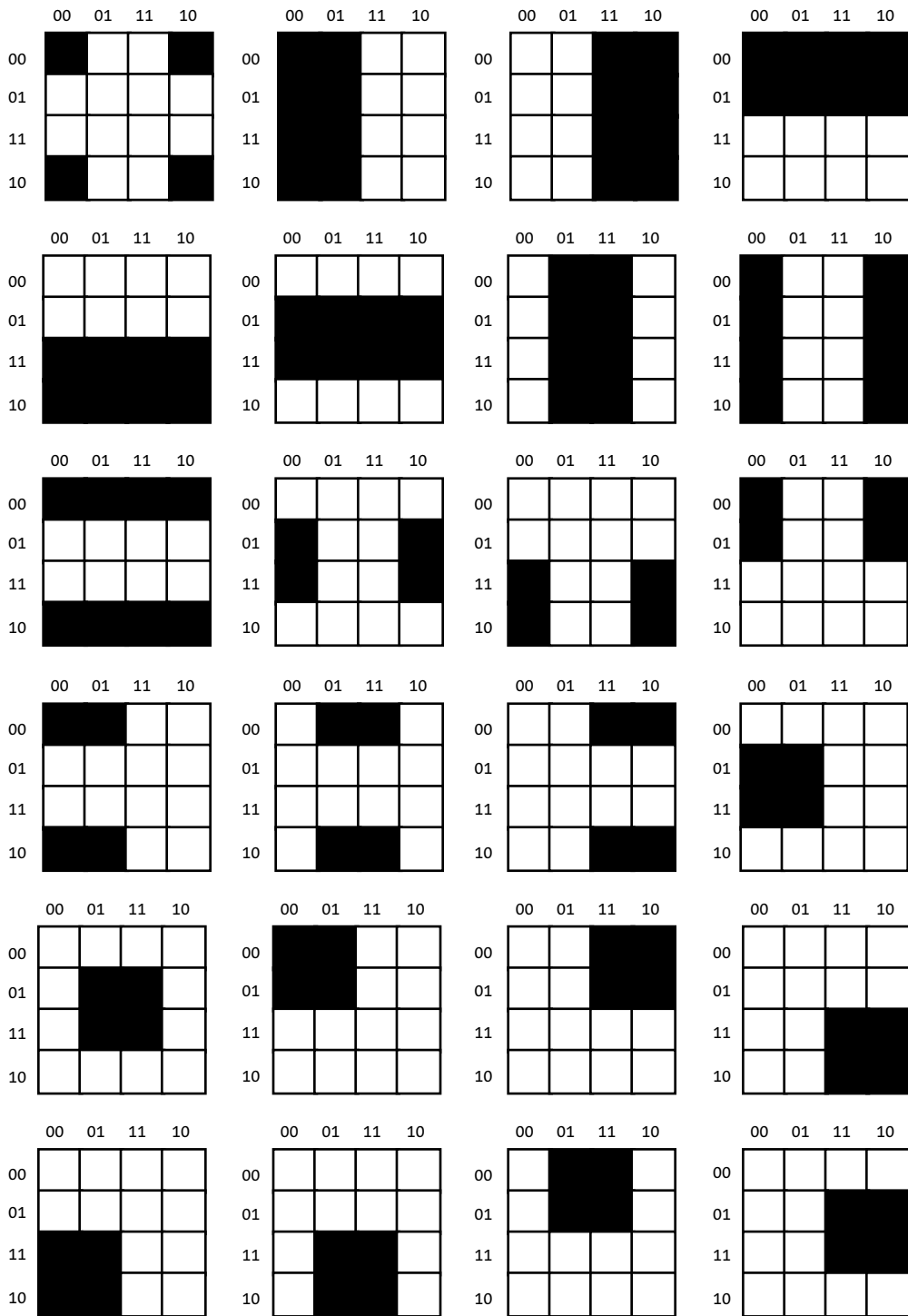


Figure 8-10: Partial pattern list for a 4x4 truth map

This final example demonstrates optimizing a function of four variables. The function is $F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CB'A' + DCB'A + DCBA + DC'B'A' + DC'BA'$, and its truth map appears in Figure 8-11.

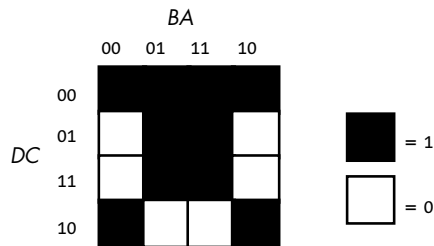
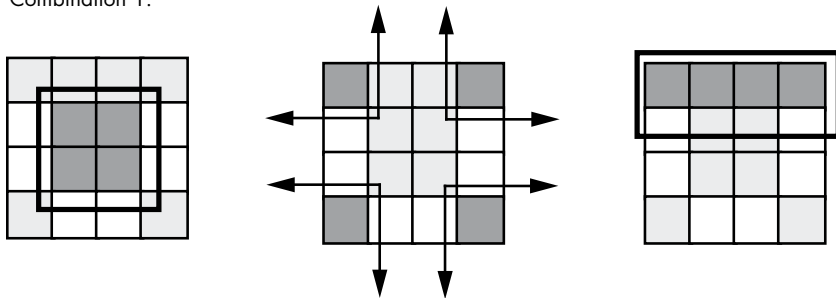


Figure 8-11: Truth map for $F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CB'A' + DCB'A + DCBA + DC'B'A' + DC'BA'$

Figure 8-12 shows the two possible sets of maximal rectangles for this function, each producing three terms.

Combination 1:



Combination 2:

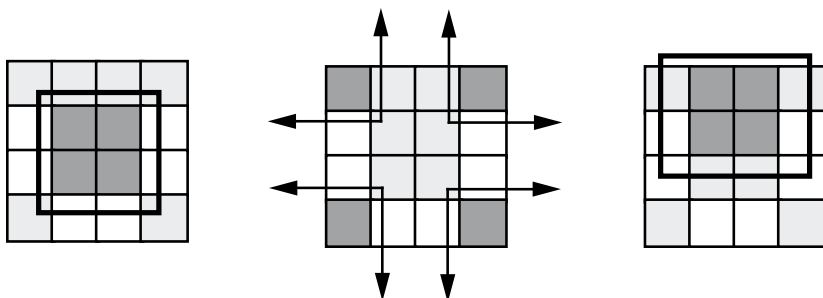


Figure 8-12: Two combinations yielding three terms

The rectangle formed by the four corners, common to both maps in Figure 8-12, contains $B, B', D,$ and D' , so we can eliminate those terms. The remaining terms contained within the rectangle are C' and A' , so this rectangle represents the term $C'A'$.

The rectangle formed by the middle four squares, also in both combinations, includes the terms $A, B, B', C, D,$ and D' . Eliminating $B, B', D,$ and D' , we obtain CA .

Combination 1 has a third term represented by the top row. This term includes the variables $A, A', B, B', C',$ and D' . We can eliminate $A, A', B,$ and B' . This leaves the term $C'D'$. Therefore, the function represented by the upper truth map is $F = CA' + CA + C'D'$.

Combination 2 has a third term represented by the top/middle four squares. This rectangle subsumes the variables $A, B, B', C, C',$ and D' . We can eliminate $B, B', C,$ and C' , leaving the term AD . Therefore, the function represented by the lower truth map is $F = CA' + CA + AD'$.

Both functions are equivalent; both are optimal (remember, there's no guarantee of a unique optimal solution). Either will suffice for our purposes: implementing Boolean functions using the fewest circuit components.

8.7 What Does This Have to Do with Computers, Anyway?

Any program you can write, you can also specify as a sequence of Boolean equations. This means that any algorithm you can implement in software, you can also implement directly in hardware—there is a one-to-one relationship between the set of all Boolean functions and the set of all electronic circuits. Electrical engineers, who design CPUs and other computer-related circuits, have to be intimately familiar with this material.

Because it's easier to specify a solution to a programming problem using languages like Pascal, C, or even assembly language than it is to specify the solution using Boolean equations, it's unlikely that you would ever implement an entire program using a set of state machines and other logic circuitry. However, a hardware solution can be orders of magnitude faster than an equivalent software solution, and some time-critical operations require a hardware solution.

It is also possible to implement all hardware functions in software. This is important, because many operations you'd normally implement in hardware are much cheaper to implement using software on a microprocessor. Indeed, one of the primary uses of assembly language on modern systems is to inexpensively replace a complex electronic circuit. Often, you can replace many tens or hundreds of dollars of electronic components with a single \$2 microcomputer chip programmed to perform the equivalent function.

The whole field of *embedded systems* (computer systems embedded in other products) deals with this problem. For example, most microwave ovens, TV sets, video games, CD players, and other consumer devices contain one or more complete computer systems whose sole purpose is to replace a complex hardware design. Engineers use computers for this purpose because they are less expensive and easier to design with than traditional electronic circuitry.

To write software that reads switches (input variables) and turns on motors, LEDs, or lights, or that locks or unlocks a door, you need to understand Boolean functions and how to implement them in software.

8.7.1 Correspondence Between Electronic Circuits and Boolean Functions

For any Boolean function, you can design an equivalent electronic circuit and vice versa. We can construct any electronic circuit using the AND, OR, and NOT Boolean operators, which correspond to the AND, OR, and inverter (NOT) circuits (see Figure 8-13). These symbols are standard electronic symbols appearing in *schematic diagrams*. (To learn more about electronic schematic diagrams, check out any book on electronic design.)

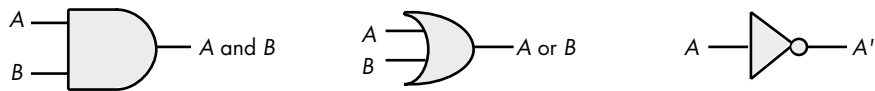


Figure 8-13: AND, OR, and inverter (NOT) gates

The lines to the left of each gate, with the *A* and *B* labels, correspond to a logic function input; the line to the right of each diagram corresponds to the function's output.

An *electronic circuit* is a combination of gates that implement some set of Boolean functions. Consider the Boolean function $F = AB + B$. You can implement this function using an AND gate and an OR gate. Simply connect the two input variables (*A* and *B*) to the inputs of the AND gate, connect the output of the AND gate to one of the inputs of the OR gate, and connect the *B* input variable to the other OR input. Now you have an electronic (hardware) circuit that implements this function.

However, you actually need only a single gate type—the NAND (NOT AND) gate—to implement *any* electronic circuit (see Figure 8-14). The NAND gate tests its two inputs (*A* and *B*) and outputs false if both inputs are true; it outputs true if both inputs are false. You could construct the NAND circuit from an AND gate and an inverter. However, from a transistor/hardware perspective, the NAND gate is actually simpler to construct than an AND gate; therefore, NAND gates (such as the 7400 IC) are very common.

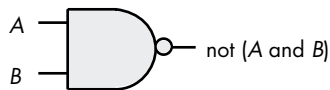


Figure 8-14: The NAND gate

We can construct any Boolean function using only NAND gates because we can build an inverter (NOT), an AND gate, and an OR gate from NAND gates.³ Building an inverter is easy; just connect the two inputs together (see Figure 8-15).

3. Indeed, the four AND gates in a 7408 Transistor-Transistor Logic (TTL) IC are probably constructed internally with transistors arranged as a NAND gate followed by an inverter.

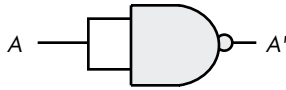


Figure 8-15: Inverter built from a NAND gate

After building an inverter, we can build an AND gate by inverting the output of a NAND gate, because NOT (NOT (A AND B)) is equivalent to A AND B (see Figure 8-16). It takes two NAND gates to construct a single AND gate (no one said that circuits constructed only with NAND gates are optimal, only that they're possible).

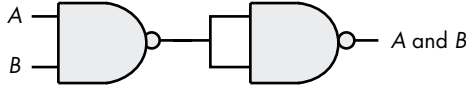


Figure 8-16: Constructing an AND gate from two NAND gates

The remaining gate is the logical-OR gate. We can construct an OR gate from NAND gates by applying DeMorgan's Theorems.

$(A \text{ or } B)'$	=	$A' \text{ and } B'$	DeMorgan's Theorem.
$A \text{ or } B$	=	$(A' \text{ and } B')'$	Invert both sides of the equation.
$A \text{ or } B$	=	$A' \text{ nand } B'$	Definition of NAND operation.

Applying these transformations produces the circuit shown in Figure 8-17.

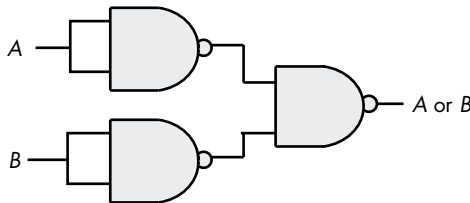


Figure 8-17: Constructing an OR gate from NAND gates

NAND gates are generally less expensive to build than other gates, and it's much easier to build up complex circuits from the same basic building blocks than it is to construct an integrated circuit using different basic gates.

8.7.2 Combinatorial Circuits

A computer's CPU is built from *combinatorial circuits*, which are systems containing basic Boolean operations (AND, OR, NOT), some inputs, and a set of outputs. A combinatorial circuit often implements several different Boolean functions, with each output corresponding to an individual logic function.

NOTE

It is very important that you remember that each output represents a different Boolean function.

8.7.2.1 Combining Addition Circuits

You can implement addition using Boolean functions. Suppose you have two 1-bit numbers, A and B . You can produce the 1-bit sum and the 1-bit carry of this addition using these two Boolean functions:

$S = AB' + A'B$	Sum of A and B .
$C = AB$	Carry from addition of A and B .

These two Boolean functions implement a *half adder*, so called because it adds 2 bits together but cannot add in a carry from a previous operation. Note that $S = 1$ if A or B is 1, $S = 0$ if A and B are both 0 or 1 (both 1 produces a carry, which is what the $C = AB$ expression produces).

A *full adder* adds three 1-bit inputs (2 bits plus a carry from a previous addition) and produces two outputs: the sum and the carry. These are the two logic equations for a full adder:

S	$= A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$
C_{out}	$= AB + AC_{in} + BC_{in}$

Although these equations produce only a single-bit result (plus a carry), it's easy to construct an n -bit sum by combining adder circuits (see Figure 8-18).

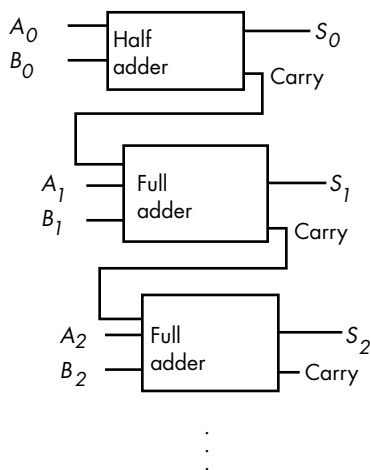


Figure 8-18: Building an n -bit adder using half and full adders

The two n -bit inputs, A and B , are passed into the adder bit-by-bit, with the LO bits input as A_0 and B_0 , and so on up to HO bits A_{n-1} and B_{n-1} . S_0 is the LO bit of the sum, up to S_{n-1} , and the final carry indicates whether the addition overflowed n bits.

8.7.2.2 Using Seven-Segment LED Decoders

Another common combinatorial circuit is the *seven-segment decoder*. Among the more important circuits in computer system design, decoder circuits enable the computer to recognize (or *decode*) a string of bits.

The seven-segment decoder circuit accepts an input of 4 bits and determines which segments to illuminate on a seven-segment LED display. Because a seven-segment display contains seven output values (one for each segment), there are seven logic functions associated with it (segments 0 through 6). See Figure 8-19 for the segment assignments. Figure 8-20 shows the active segments for each of the 10 decimal values.

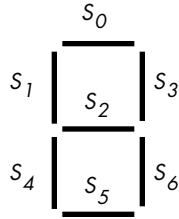


Figure 8-19: Seven-segment display

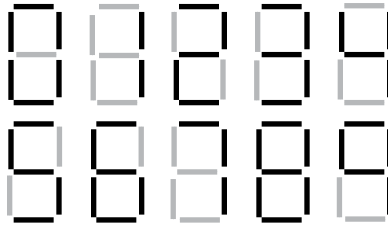


Figure 8-20: Seven-segment values for 0 through 9

The four inputs to each of these seven Boolean functions are the 4 bits from a binary number in the range 0 through 9. Let D be the HO bit of this number and A be the LO bit. Each segment's logic function should produce a 1 (segment on) for all binary number inputs that have that segment illuminated in Figure 8-20. For example, S_4 (segment 4) should be illuminated for numbers 0, 2, 6, and 8, which correspond to the binary values 0000, 0010, 0110, and 1000. For each of the binary values that illuminates a segment, you will have one minterm in the logic equation:

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'$$

S_0 (segment 0), as a second example, is on for the numbers 0, 2, 3, 5, 6, 7, 8, and 9, which correspond to the binary values 0000, 0010, 0011, 0101, 0110, 0111, 1000, and 1001. Therefore, the logic function for S_0 is as follows:

$$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A$$

8.7.2.3 Decoding Memory Addresses

A decoder is also commonly used in memory expansion. For example, suppose a system designer wishes to install four (identical) 256MB memory modules in a system to bring the total to 1GB of RAM. Each of these 256MB memory modules has 28 address lines ($A_0..A_{27}$), assuming each memory module is 8 bits wide ($2^{28} \times 8$ bits is 256MB).⁴

Unfortunately, if the system designer hooked up those four memory modules to the CPU's address bus, each module would respond to the same addresses on the bus. Pandemonium would result. To correct this problem, each memory module needs to respond to a different set of addresses appearing on the full address bus (with a module address appearing on the LO 28 bits of the address bus). By adding a chip-select line to each of the memory modules, and using a two-input, four-output decoder circuit, we can use the chip select lines A_{28} and A_{29} to specify the HO 2 bits of the (now effectively 30-bit) memory address. See Figure 8-21 for the details.

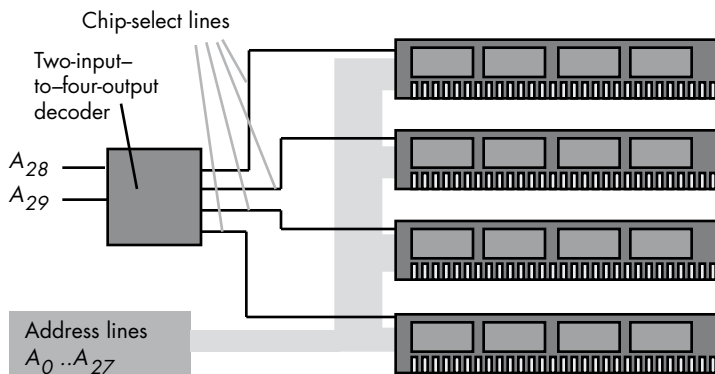


Figure 8-21: Adding four 256MB memory modules to a system

The two- to four-line decoder circuit in Figure 8-21 incorporates four different logic functions: one function for each of the outputs. Each combination of the input bits will activate a single chip-select line, and deactivate the other three. Assuming the inputs are A and B ($A = A_{28}$ and $B = A_{29}$), the four output functions are as follows:

$$\begin{aligned}Q_0 &= A'B' \\Q_1 &= AB' \\Q_2 &= A'B \\Q_3 &= AB\end{aligned}$$

Following standard electronic circuit notation, these equations use Q to denote an output.

Note that most circuit designers use *active low logic* for decoders and chip enables. This means that they enable a circuit when a low-input

4. Actually, most memory modules are wider than 8 bits, so a real 256MB memory module will have fewer than 28 address lines, but we'll ignore this technicality in this example.

value (0) is supplied and disable the circuit when a high-input value (1) is supplied. Real-world decoding circuits would likely use the following sums of maxterms functions:

$$\begin{aligned} Q_0 &= A + B \\ Q_1 &= A' + B \\ Q_2 &= A + B' \\ Q_3 &= A' + B' \end{aligned}$$

8.7.2.4 Decoding Machine Instructions

Decoding circuits are also used to decode machine instructions. We’ll cover this subject in much greater depth in Chapters 9 and 10, but a simple example is in order here.

Most modern computer systems represent machine instructions using binary values in memory. To execute an instruction, the CPU fetches the instruction’s binary value from memory, decodes it using decoder circuitry, and then does the appropriate work. To see how this is done, let’s create a fictional CPU with a very simple instruction set. Figure 8-22 provides the instruction format (all the numeric codes that correspond to the various instructions) for our CPU. Within the 1-byte operation code (opcode), 3 bits (iii) represent the instruction, 2 bits (ss) the source operand, and 2 bits the destination operand (dd).

Bit:	7	6	5	4	3	2	1	0
	0	i	i	i	s	s	d	d

iii		ss & dd	
000 =	mov	00 =	eax
001 =	add	01 =	ebx
010 =	sub	10 =	ecx
011 =	mul	11 =	edx
100 =	div		
101 =	and		
110 =	or		
111 =	xor		

Figure 8-22: Instruction (opcode) format for a very simple CPU

To determine the 8-bit opcode for a given instruction, look up each component of the instruction in the tables in Figure 8-22 and substitute the corresponding bit values.

Let’s pick `mov(eax, ebx);` as our simple example. To convert this instruction to its numeric equivalent, `mov` is encoded as 000, `eax` is encoded as 00, and `ebx` is encoded as 01. Assemble these three fields into the opcode byte (a packed data type), to obtain the bit value: `%00000001`. Therefore, the numeric value \$1 is the value for the `mov(eax, ebx);` instruction (see Figure 8-23).

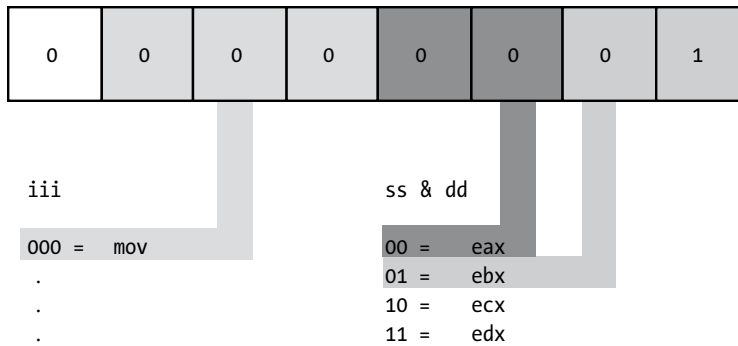


Figure 8-23: Encoding the `mov(eax, ebx)` instruction

A typical decoder circuit for this example appears in Figure 8-24. The circuit uses three separate decoders to decode the individual fields of the opcode. This is much less complex than creating a single 7- to 128-line decoder to decode the entire opcode.

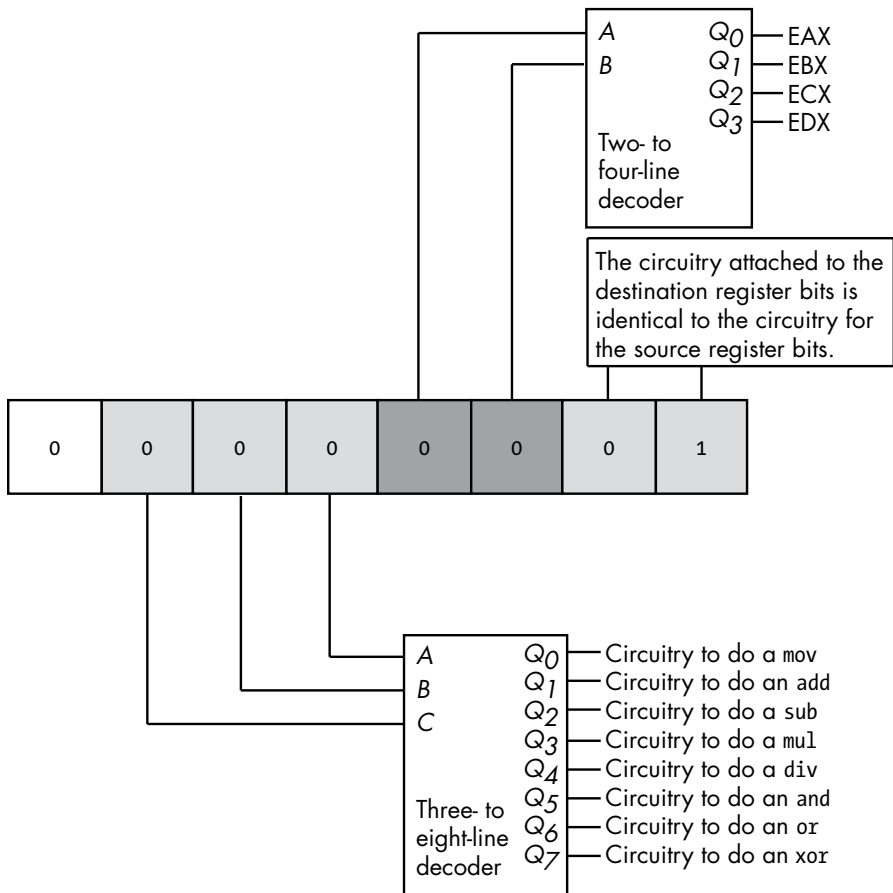


Figure 8-24: Decoding simple machine instructions

The circuit in Figure 8-24 tells you which instruction and what operands a given opcode specifies. To actually execute this instruction, you must supply additional circuitry to select the source and destination operands from an array of registers and act accordingly upon those operands. Such circuitry is beyond the scope of this chapter, so we'll save the juicy details for later.

8.7.3 Sequential and Clocked Logic

One major problem with combinatorial logic is that it is *memoryless*. In theory, all logic function outputs depend only on the current inputs. Any change in the input values immediately appears on the outputs.⁵ Unfortunately, computers need the ability to *remember* the results of past computations. This is the domain of sequential, or clocked, logic.

8.7.3.1 The Set/Reset Flip-Flop

A *memory cell* is an electronic circuit that remembers an input value after the removal of that input value. The most basic memory unit is the *set/reset (S/R) flip-flop*. You can construct an S/R flip-flop memory cell using two NAND gates, as shown in Figure 8-25. In this diagram, the outputs of the two NAND gates are recirculated to one of the inputs of the other NAND gate.

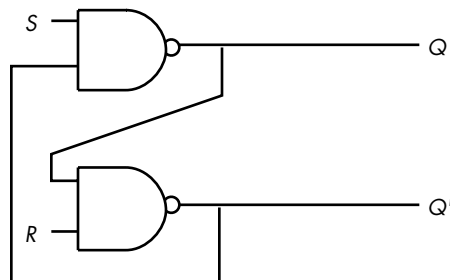


Figure 8-25: Set/reset flip-flop constructed from NAND gates

The *S* and *R* inputs are normally high, or 1. If you toggle the *S* input by temporarily setting its value to 0 and then bringing it back to 1, the *Q* output is set to 1. Likewise, if you toggle the *R* input from 1 to 0 and back to 1, this resets the *Q* output to 0. *Q'* outputs the opposite of *Q*.

If both *S* and *R* are 1, then the *Q* output depends upon the original value of *Q* itself. That is, whatever *Q* happens to be, the top NAND gate continues to output that same value. If *Q* was originally 1, then the bottom NAND gate receives two inputs of 1 (both *Q* and *R*), and the bottom NAND gate produces an output of 0 (*Q'*). As a result, the two inputs to the top NAND gate are 0 and 1, and the top NAND gate produces an output of 1, matching the original value for *Q*.

5. In practice, there is a short *propagation delay* between a change in the inputs and the corresponding outputs in any electronic implementation of a Boolean function.

On the other hand, if the original value of Q was 0, then the inputs to the bottom NAND gate are $Q = 0$ and $R = 1$, and the output of this bottom NAND gate is 1. As a result, the inputs to the top NAND gate are $S = 1$ and $Q' = 1$. This produces a 0 output, the original value of Q .

Now suppose Q is 0, S is 0, and R is 1. This sets the two inputs to the top NAND gate to 1 and 0, forcing the output (Q) to 1. Returning S to the high state does not change the output at all, because the value of Q' is 1. You will obtain this same result if Q is 1, S is 0, and R is 1. Again, this produces a Q output value of 1, and again this value remains 1 even when S switches from 0 to 1. To overcome this and produce a Q output of 1, you must toggle the S input. The same idea applies to the R input, except that toggling it forces the Q output to 0 rather than to 1.

There is one catch to this circuit. It does not operate properly if you set both the S and R inputs to 0 simultaneously. This forces both the Q and Q' outputs to 1 (which is logically inconsistent). Whichever input remains 0 the longest determines the final state of the flip-flop. A flip-flop operating in this mode is said to be *unstable*.

Table 8-9 lists all the output configurations for an S/R flip-flop based on the current inputs and the previous output values.

Table 8-9: S/R Flip-Flop Output States Based on Current Inputs and Previous Outputs

Previous Q	Previous Q'	S input	R input	Q output	Q' output
x^6	x	0 ($1 > 0 > 1$)	1	1	0
x	x	1	0 ($1 > 0 > 1$)	0	1
x	x	0	0	1	1^7
0	1	1	1	0	1
1	0	1	1	1	0

8.7.3.2 The D Flip-Flop

The only problem with the S/R flip-flop is that to be able to remember either a 0 or a 1 value, you must have two different inputs. A memory cell would be more valuable to us if we could specify the data value to remember with one input value and supply a second *clock input* value to *latch* the data input value.⁸ This type of flip-flop, the D flip-flop (D stands for *data*), uses the circuit in Figure 8-26.

6. x = “don’t care,” implying that the value may be 0 or 1 and it won’t affect the outputs.

7. This is an *unstable* configuration and will change once S or R is set to 1.

8. “Latch” simply means to *remember* the value. That is, a D flip-flop is the basic memory element because it can remember one data bit appearing on its D input.

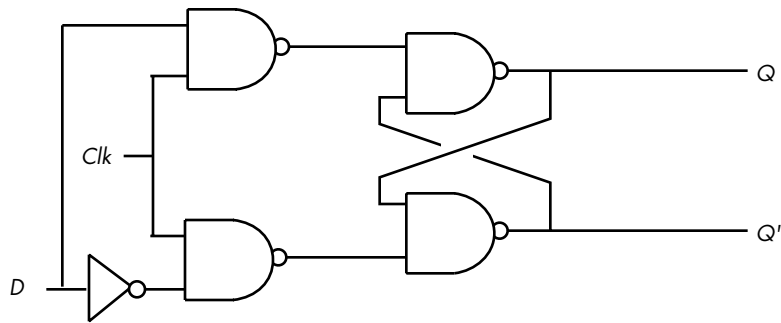


Figure 8-26: Implementing a D flip-flop with NAND gates

Assuming you fix the Q and Q' outputs to either 0/1 or 1/0, sending a *clock pulse* that goes from 0 to 1 and back to 0 will copy the D input to the Q output (and set Q' to the inverse of Q). To see how this works, note that the right half of the circuit diagram in Figure 8-26 is an S/R flip-flop. If the data input is 1 while the clock line is high, this places a 0 on the S input of the S/R flip-flop (and a 1 on the R input). Conversely, if the data input is 0 while the clock line is high, this places a 0 on the R input (and a 1 on the S input) of the S/R flip-flop, thus clearing the S/R flip-flop's output. Whenever the clock input is low, both the S and R input are high, and the outputs of the S/R flip-flop do not change.

Although remembering a single bit is often important, in most computer systems you want to remember a *group* of bits. You can do this by combining several D flip-flops in parallel. Concatenating flip-flops to store an n -bit value forms a *register*. The electronic schematic in Figure 8-27 shows how to build an 8-bit register from a set of D flip-flops.

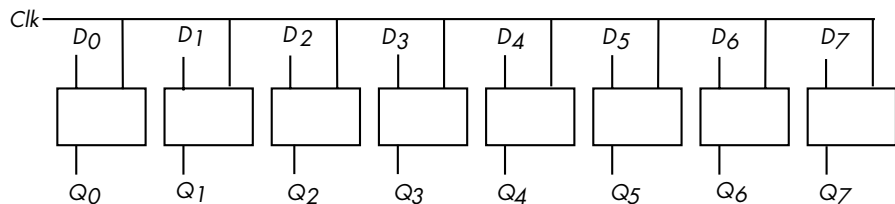


Figure 8-27: An 8-bit register implemented with eight D flip-flops

Note that the eight D flip-flops in Figure 8-27 use a common clock line. This diagram does not show the Q' outputs on the flip-flops because they are rarely required in a register.

D flip-flops are useful for building many sequential circuits beyond simple registers. For example, you can build a *shift register* that shifts the bits one position to the left on each clock pulse. A 4-bit shift register appears in Figure 8-28.

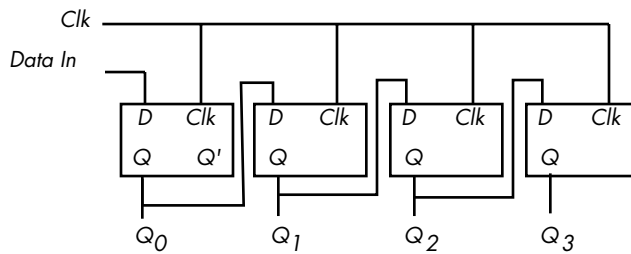


Figure 8-28: A 4-bit shift register built from D flip-flops

You can even build a *counter* that counts the number of times the clock toggles from 1 to 0 and back to 1 using flip-flops. The circuit in Figure 8-29 implements a 4-bit counter using D flip-flops.

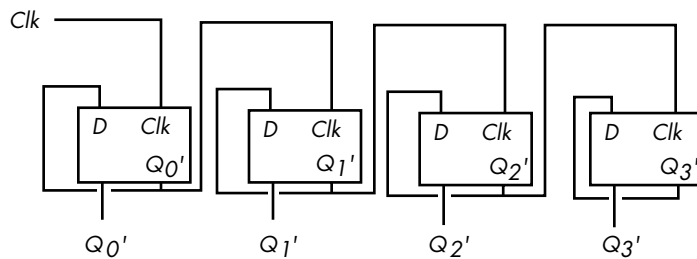


Figure 8-29: A 4-bit counter built from D flip-flops

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits. For example, you can build a simple state machine known as a *sequencer* by combining a counter and a decoder, as shown in Figure 8-30.

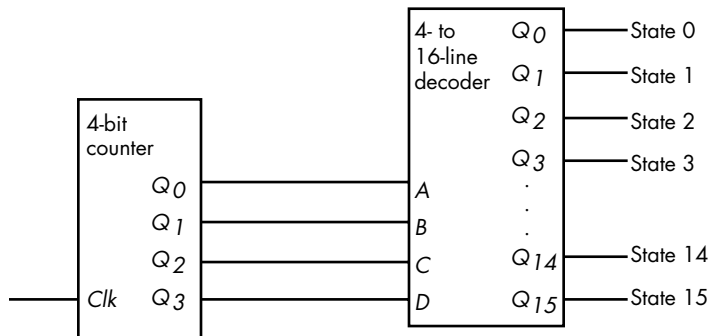


Figure 8-30: A simple 16-state sequencer

For each cycle of the clock in Figure 8-30, this sequencer activates one of its output lines. Those lines, in turn, may control other circuits. By “firing” those other circuits on each of the 16 output lines of the decoder, we can control the order in which the circuits accomplish their tasks. This is essential in a CPU, as we often need to control the sequence of various

operations. For example, it wouldn't be a good thing if the `add(eax, ebx);` instruction stored the result into EBX before fetching the source operand from EAX (or EBX). A simple sequencer can tell the CPU when to fetch the first operand, when to fetch the second operand, when to add them together, and when to store the result. However, we're getting a little ahead of ourselves—we'll discuss this in detail in the next two chapters.

8.8 For More Information

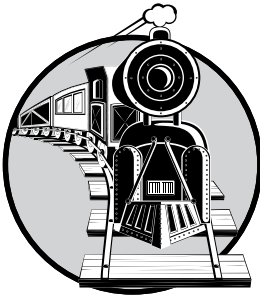
Horowitz, Paul, and Winfield Hill. *The Art of Electronics*. 3rd ed. Cambridge, UK: Cambridge University Press, 2015.

NOTE

This chapter is not, by any means, a complete treatment of Boolean algebra and digital design. If you're interested in learning more, consult one of the dozens of books on this subject.

9

CPU ARCHITECTURE



Without question, the design of the central processing unit (CPU) has the greatest impact on the performance of your software. To execute a particular instruction (or command), a CPU requires a certain amount of electronic circuitry specific to that instruction. As you increase the number of instructions the CPU can support, you also increase the CPU's complexity and the amount of circuitry, or *logic gates*, needed to execute them. Therefore, to keep the number of logic gates and the associated costs reasonably small, CPU designers must restrict the number and complexity of the instructions the CPU can execute. This is known as the CPU's *instruction set*.

This chapter, and the next, discusses the design of CPUs and their instruction sets—information that is absolutely crucial for writing high-performance software.

9.1 Basic CPU Design

Programs in early computer systems were often hardwired into the circuitry. That is, the computer's wiring determined exactly what algorithm the computer would execute. The computer had to be rewired in order to solve a different problem. This was a difficult task, something that only electrical engineers were able to do.

Thus, the next advance in computer design was the programmable computer system, in which a computer operator could easily “rewire” the computer using a panel of sockets and plug wires known as a *patch board*. A computer program consisted of rows of sockets, with each row representing one operation (instruction) during the program's execution. To execute an instruction, the programmer inserted a wire into its corresponding socket (see Figure 9-1).

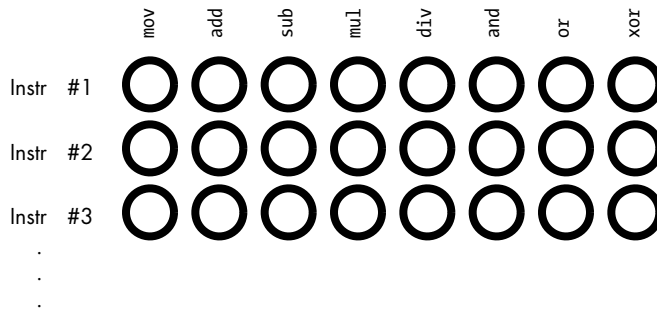


Figure 9-1: Patch board programming

The number of possible instructions was limited by how many sockets could fit on each row. CPU designers quickly realized that with a small amount of additional logic circuitry, they could reduce the number of sockets required for specifying n different instructions from n sockets to $\log_2(n)$ sockets. They did this by assigning a unique binary number to each instruction (for example, Figure 9-2 shows how to represent eight instructions using only 3 bits).

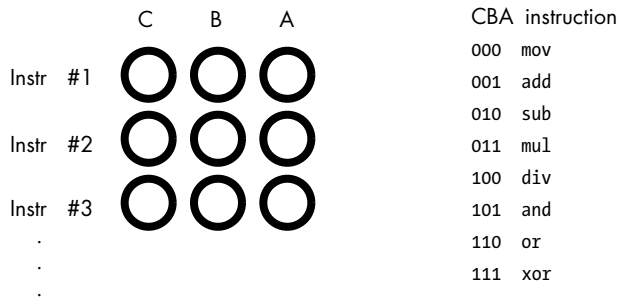


Figure 9-2: Encoding instructions

The example in Figure 9-2 requires eight logic functions to decode the *A*, *B*, and *C* bits on the patch board, but the extra circuitry (a single three-to-eight-line decoder) is worth the cost, because it reduces the total number of sockets from eight to three for each instruction.

Many CPU instructions require operands. For example, the *mov* instruction moves data from one location in the computer to another, such as from one register to another, and therefore requires a source operand and a destination operand. The operands were encoded as part of the machine instruction, with sockets corresponding to the source and destination. Figure 9-3 shows one possible combination of sockets to handle a *mov* instruction.

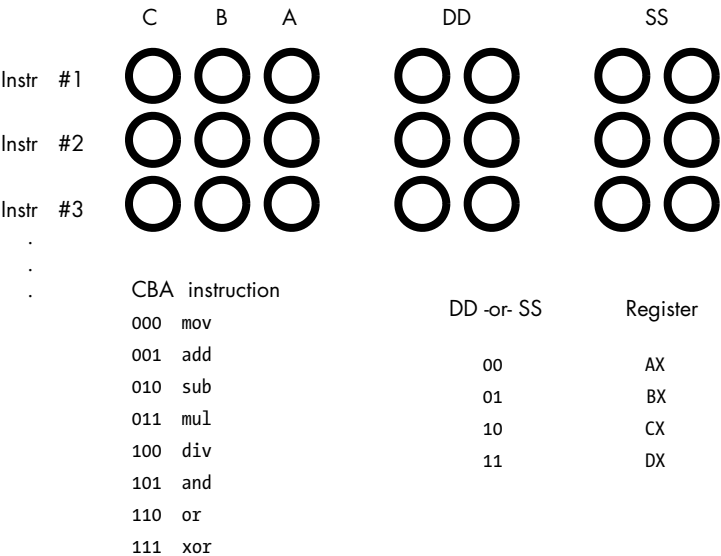


Figure 9-3: Encoding instructions with source and destination fields

The *mov* instruction would move data from the source register to the destination register, the *add* instruction would add the value of the source register to the destination register, and so on. This scheme allowed the encoding of 128 different instructions with just seven sockets per instruction.

As noted earlier, a big problem with patch-board programming was that a program’s functionality was limited by the number of sockets available on the machine. Early computer designers recognized a relationship between the sockets on the patch board and bits in memory. They realized they could store the binary equivalent of a machine instruction in main memory, fetch that binary number when the CPU wanted to execute the instruction, and then load it into a special register to decode the instruction. Known as the *stored program computer*, this invention was another major advance in computer design.

The trick was to add more circuitry, called the *control unit (CU)*, to the CPU. The control unit uses a special register, the *instruction pointer*, to hold the address of an instruction’s binary numeric code (also known as an

operation code or *opcode*). The control unit fetches the instruction's opcode from memory and places it in the instruction decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution.

9.2 Decoding and Executing Instructions: Random Logic vs. Microcode

Once the control unit fetches an instruction from memory, traditional CPUs use two common approaches to execute the instruction: random logic (hard-wired) and microcode (emulation). The 80x86 family, for example, uses both of these techniques.

The *random logic*¹ or hardwired approach uses decoders, latches, counters, and other hardware logic devices to operate on the opcode data. Random logic is fast but poses a circuitry design challenge; for CPUs with large and complex instruction sets, it's difficult to properly lay out the logic so that related circuits are close to one another in the two-dimensional space of the chip.

CPUs based on microcode contain a small, very fast *execution unit* (circuitry responsible for executing a particular function), known as a *micro-engine*, that uses the binary opcode to select a set of instructions from the microcode bank. This microcode executes one microinstruction per clock cycle, and the sequence of microinstructions executes all the steps to perform whatever calculations are necessary for that instruction.

Although this *microengine* itself is fast, it must fetch its instructions from the microcode ROM (read-only memory). Therefore, if memory technology is slower than the execution logic, the micro-engine must run at the same speed as the microcode ROM, which in turn limits the speed at which the CPU can run.

The random logic approach decreases the time to execute an opcode's instruction, provided that typical CPU speeds are faster than memory speeds, but that doesn't mean it's necessarily faster than the microcode approach. Random logic often includes a sequencer that steps through several states (one state per clock cycle). Whether you use up clock cycles executing microinstructions or stepping through a random logic state machine, you're still burning up time.

Which approach is better for CPU design depends entirely on the current state of memory technology. If memory technology is faster than CPU technology, the microcode approach probably makes more sense. If memory technology is slower than CPU technology, random logic tends to execute machine instructions more quickly.

1. There is actually nothing random about this logic at all. The name comes from the fact that if you view a photomicrograph of a CPU die that uses microcode, the microcode section looks very regular; the same photograph of a CPU that utilizes hardware logic contains no such easily discernible patterns.

9.3 Executing Instructions, Step by Step

Regardless of which approach the CPU uses, you need to understand how a CPU executes individual machine instructions. To that end, we'll consider four representative 80x86 instructions—`mov`, `add`, `loop`, and `jnz` (*jump if not zero*)—to give you a sense of how a CPU executes all the instructions in its instruction set.

As you saw earlier, the `mov` instruction copies data from a source operand to a destination operand. The `add` instruction adds the value of its source operand to its destination operand. `loop` and `jnz` are *conditional jump* instructions—they test some condition and, if it's true, they jump to some other instruction in memory; if it's false, they continue with the next instruction. The `jnz` instruction tests a Boolean variable within the CPU known as the *zero flag* and either transfers control to the target instruction (the instruction to jump to) if the zero flag contains 0, or continues with the next instruction if the zero flag contains 1. The program indicates the address of the target instruction by specifying the distance, in bytes, between it and the `jnz` instruction in memory.

The `loop` instruction decrements the value of the ECX register and, if the resulting value does not contain 0, transfers control to a target instruction. This is a good example of a *complex instruction set computer (CISC)* instruction because it does more than one operation:

1. It subtracts 1 from ECX.
2. It does a conditional jump if ECX does not contain 0.

That is, `loop` is roughly equivalent to the following instruction sequence:

```
sub( 1, ecx ); // On the 80x86, the sub instruction sets the zero flag
jnz SomeLabel; // the result of the subtraction is 0.
```

To execute the `mov`, `add`, `jnz`, and `loop` instructions, the CPU has to execute a number of different operations. Each operation requires a finite amount of time to execute, and the time required to execute the entire instruction generally amounts to one clock cycle per operation or *stage* (step) that the CPU executes. Obviously, the more stages needed for an instruction, the slower it will run. Because they have many execution stages, complex instructions generally run slower than simple instructions.

Although 80x86 CPUs differ and don't necessarily execute the exact same steps, their sequence of operations is similar. This section presents some possible sequences, all starting with the same three execution stages:

1. Fetch the instruction's opcode from memory.
2. Update the EIP (extended instruction pointer) register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.

9.3.1 The *mov* Instruction

A decoded 32-bit 80x86 `mov(srcReg, destReg);` instruction might use the following (additional) execution stages:

1. Fetch the data from the source register (*srcReg*).
2. Store the fetched value into the destination register (*destReg*).

The `mov(srcReg, destMem);` instruction could use the following execution stages:

1. Fetch the displacement associated with the memory operand from the memory location immediately following the opcode.
2. Update EIP to point at the first byte beyond the operand that follows the opcode.
3. Compute the effective address of the destination memory location, if the `mov` instruction uses a complex addressing mode (for example, the indexed addressing mode).
4. Fetch the data from *srcReg*.
5. Store the fetched value into the destination memory location.

A `mov(srcMem, destReg);` instruction is very similar, simply swapping the register access for the memory access in these steps.

The `mov(constant, destReg);` instruction could use the following execution stages:

1. Fetch the constant associated with the source operand from the memory location immediately following the opcode.
2. Update EIP to point at the first byte beyond the constant that follows the opcode.
3. Store the constant value into the destination register.

Assuming each stage requires one clock cycle for execution, this sequence (including the three common stages) will require six clock cycles to execute.

The `mov(constant, destMem);` instruction could use the following execution stages:

1. Fetch the displacement associated with the memory operand from the memory location immediately following the opcode.
2. Update EIP to point at the first byte beyond the operand that follows the opcode.
3. Fetch the constant operand's value from the memory location immediately following the displacement associated with the memory operand.
4. Update EIP to point at the first byte beyond the constant.

5. Compute the effective address of the destination memory location, if the `mov` instruction uses a complex addressing mode (for example, the indexed addressing mode).
6. Store the constant value into the destination memory location.

9.3.2 The `add` Instruction

The `add` instruction is a little more complex. Here's a typical set of operations (beyond the common set) that the decoded `add(srcReg, destReg);` instruction must complete:

1. Fetch the value of the source register and send it to the *arithmetic logical unit (ALU)*, which handles arithmetic on the CPU.
2. Fetch the value of the destination register operand and send it to the ALU.
3. Instruct the ALU to add the values.
4. Store the result back into the destination register operand.
5. Update the flags register with the result of the addition operation.

NOTE

The flags register, also known as the condition-codes register or program-status word, is an array of Boolean variables in the CPU that tracks whether the previous instruction produced an overflow, a zero result, a negative result, or other such condition.

If the source operand is a memory location instead of a register, and the `add` instruction takes the form `add(srcMem, destReg);`, then the instruction sequence is slightly more complicated:

1. Fetch the displacement associated with the memory operand from the memory location immediately following the opcode.
2. Update EIP to point at the first byte beyond the operand that follows the opcode.
3. Compute the effective address of the source memory location, if the `add` instruction uses a complex addressing mode (for example, the indexed addressing mode).
4. Fetch the source operand's data from memory and send it to the ALU.
5. Fetch the value of the destination register operand and send it to the ALU.
6. Instruct the ALU to add the values.
7. Store the result back into the destination register operand.
8. Update the flags register with the result of the addition operation.

If the source operand is a constant and the destination operand is a register, the `add` instruction takes the form `add(constant, destReg);` and the CPU might deal with it as follows:

1. Fetch the constant operand that immediately follows the opcode in memory and send it to the ALU.

2. Update EIP to point at the first byte beyond the constant that follows the opcode.
3. Fetch the value of the destination register operand and send it to the ALU.
4. Instruct the ALU to add the values.
5. Store the result back into the destination register operand.
6. Update the flags register with the result of the addition operation.

This instruction sequence requires nine cycles to complete.

If the source operand is a constant, and the destination operand is a memory location, then the `add` instruction takes the form `add(constant, destMem)`; and the sequence is slightly more complicated:

1. Fetch the displacement associated with the memory operand from memory immediately following the opcode.
2. Update EIP to point at the first byte beyond the operand that follows the opcode.
3. Compute the effective address of the destination memory location, if the `add` instruction uses a complex addressing mode (for example, the indexed addressing mode).
4. Fetch the constant operand that immediately follows the memory operand's displacement value and send it to the ALU.
5. Fetch the destination operand's data from memory and send it to the ALU.
6. Update EIP to point at the first byte beyond the constant that follows the memory operand.
7. Instruct the ALU to add the values.
8. Store the result back into the destination memory operand.
9. Update the flags register with the result of the addition operation.

This instruction sequence requires 11 or 12 cycles to complete, depending on whether the effective address computation is necessary.

9.3.3 The *jnz* Instruction

Because the 80x86 `jnz` instruction does not allow different types of operands, it needs only one sequence of steps. The `jnz label;` instruction might use the following additional execution stages once decoded:

1. Fetch the displacement value (the jump distance) and send it to the ALU.
2. Update the EIP register to hold the address of the instruction following the displacement operand.
3. Test the zero flag to see if it is clear (that is, if it contains 0).
4. If the zero flag was clear, copy the value in EIP to the ALU.
5. If the zero flag was clear, instruct the ALU to add the displacement and EIP values.
6. If the zero flag was clear, copy the result of the addition back to the EIP.

Notice how the `jnz` instruction requires fewer steps, and thus runs in fewer clock cycles, if the jump is not taken. This is very typical for conditional jump instructions.

9.3.4 The *loop* Instruction

Because the 80x86 `loop` instruction does not allow different types of operands, it needs only one sequence of steps. The decoded 80x86 `loop` instruction might use an execution sequence like the following:²

1. Fetch the value of the ECX register and send it to the ALU.
2. Instruct the ALU to decrement this value.
3. Send the result back to the ECX register. Set a special internal flag if this result is nonzero.
4. Fetch the displacement value (the jump distance) following the opcode in memory and send it to the ALU.
5. Update the EIP register with the address of the instruction following the displacement operand.
6. Test the special internal flag to see if ECX was nonzero.
7. If the flag was set (that is, it contains 1), copy the value in EIP to the ALU.
8. If the flag was set, instruct the ALU to add the displacement and EIP values.
9. If the flag was set, copy the result of the addition back to the EIP register.

As with the `jnz` instruction, note that the `loop` instruction executes more rapidly if the branch is not taken, and the CPU continues execution with the instruction that immediately follows the `loop` instruction.

9.4 RISC vs. CISC: Improving Performance by Executing More, Faster, Instructions

Early microprocessors (including the 80x86 and its predecessors) are examples of *complex instruction set computers (CISCs)*. At the time these CPUs were created, the thinking was that having each instruction do more work made programs run faster because they executed fewer instructions (as CPUs with less complex instructions had to execute more instructions to do the same amount of work). The Digital Equipment Corporation (DEC) PDP-11 and its successor, the VAX, epitomized this design philosophy.

In the early 1980s, computer architecture researchers discovered that this complexity came at a huge cost. All the hardware necessary to support these complex instructions wound up constraining the overall clock speed of the CPU. Experiments with the VAX 11-780 minicomputer demonstrated that programs executing multiple, simple, instructions were faster than those executing fewer, more complex, instructions. Those researchers

2. Plus, of course, the common instructions at the beginning of the sequence.

hypothesized that if they stripped the instruction set down to the bare essentials, using only simple instructions, they could boost the hardware's performance (by increasing the clock speed). They called this new architecture *reduced instruction set computer (RISC)*.³ So began the great “RISC versus CISC” debate: which architecture was really better?

On paper, at least, RISC CPUs looked better. In practice, they ran at slower clock speeds, because existing CISC designs had a huge head start (as their designers had had many more years to optimize them). By the time RISC CPU designs had matured enough to run at higher clock speeds, the CISC designs had evolved, taking advantage of the RISC research. Today, the 80x86 CISC CPU is still the high-performance king. RISC CPUs found a different niche: they tend to be more power efficient than CISC processors, so they typically wind up in portable and low-power designs (such as cell phones and tablets).

Though the 80x86 (a CISC CPU) remains the performance leader, it's still possible to write programs with a larger number of simple 80x86 instructions that run faster than those with fewer, more complex 80x86 instructions. 80x86 designers have kept these legacy instructions around to allow you to execute older software that still contains them. Newer compilers, however, avoid these legacy instructions to produce faster-running code.

Nevertheless, one important takeaway from RISC research is that the execution time of each instruction is largely dependent upon the amount of work it does. The more internal operations an instruction requires, the longer it will take to execute. In addition to improving execution time by reducing the number of internal operations, RISC also prioritized internal operations that could execute concurrently—that is, *in parallel*.

9.5 Parallelism: The Key to Faster Processing

If we can reduce the amount of time it takes for a CPU to execute the individual instructions in its instruction set, an application containing a sequence of those instructions will also run faster than it otherwise would.

An early goal of the RISC processors was to execute one instruction per clock cycle, on average. However, even if a RISC instruction is simplified, its actual execution still requires multiple steps. So how could the processors achieve this goal? The answer is parallelism.

Consider the following steps for a `mov(srcReg, destReg);` instruction:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the data from `srcReg`.
5. Store the fetched value into the destination register (`destReg`).

3. Note, by the way, that RISC should be read as “(reduced instruction) set computer,” not “reduced (instruction set) computer.” RISC reduces the complexity of each instruction, not the size of the instruction set.

The CPU must fetch the instruction's opcode from memory before it updates the EIP register instruction with the address of the byte beyond the opcode, decode the opcode before it knows to fetch the value of the source register, and fetch the value of the source register before it can store the fetched value in the destination register.

All but one of the stages in the execution of this `mov` instruction are *serial*. That is, the CPU must execute one stage before proceeding to the next. The exception is step 2, updating the EIP register. Although this stage must follow the first stage, none of the following stages depend upon it. We could execute this step concurrently with any of the others, and it wouldn't affect the operation of the `mov` instruction. By doing two of the stages in parallel, then, we can reduce this instruction's execution time by one clock cycle. The following sequence illustrates one possible concurrent execution:

1. Fetch the instruction's opcode from memory.
2. Decode the instruction's opcode to see what instruction it specifies.
3. Fetch the data from *srcReg* and update the EIP register with the address of the byte following the opcode.
4. Store the fetched value into the destination register (*destReg*).

Although the remaining stages in the `mov(srcReg, destReg);` instruction must be serialized, other forms of the `mov` instruction offer similar opportunities to save cycles by executing stages concurrently. For example, consider the 80x86 `mov([ebx+disp], eax);` instruction:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement value for use in calculating the effective address of the source operand.
5. Update EIP to point at the first byte after the displacement value in memory.
6. Compute the effective address of the source operand.
7. Fetch the value of the source operand's data from memory.
8. Store the result into the destination register operand.

Once again, we can overlap the execution of several stages in this instruction. In the following example, we reduce the number of steps from eight to six by overlapping both updates of EIP with two other operations:

1. Fetch the instruction's opcode from memory.
2. Decode the instruction's opcode to see what instruction it specifies, and update the EIP register with the address of the byte following the opcode.
3. Fetch the displacement value for use in calculating the effective address of the source operand.

4. Compute the effective address of the source operand, *and* update EIP to point at the first byte after the displacement value in memory.
5. Fetch the value of the source operand's data from memory.
6. Store the result into the destination register operand.

As a last example, consider the `add(constant, [ebx+disp]);` instruction. Its serial execution looks like this:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register with the address of the byte following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement value from the memory location immediately following the opcode.
5. Update EIP to point at the first byte beyond the displacement operand that follows the opcode.
6. Compute the effective address of the second operand.
7. Fetch the constant operand that immediately follows the displacement value in memory and send it to the ALU.
8. Fetch the destination operand's data from memory and send it to the ALU.
9. Update EIP to point at the first byte beyond the constant that follows the displacement operand.
10. Instruct the ALU to add the values.
11. Store the result back into the destination (second) operand.
12. Update the flags register with the result of the addition operation.

We can overlap several stages in this instruction because they don't depend on the result of their immediate predecessor:

1. Fetch the instruction's opcode from memory.
2. Decode the instruction's opcode to see what instruction it specifies *and* update the EIP register with the address of the byte following the opcode.
3. Fetch the displacement value from the memory location immediately following the opcode.
4. Update EIP to point at the first byte beyond the displacement operand that follows the opcode *and* compute the effective address of the memory operand (`ebx+disp`).
5. Fetch the constant operand that immediately follows the displacement value and send it to the ALU.
6. Fetch the destination operand's data from memory and send it to the ALU.
7. Instruct the ALU to add the values *and* update EIP to point at the first byte beyond the constant value.
8. Store the result back into the second operand *and* update the flags register with the result of the addition operation.

Although it might seem like the CPU could fetch the constant and the memory operand in the same stage because their values do not depend upon each other, it can't do this (yet!) because it has only a single data bus, and both values are coming from memory. In the next section you'll see how we can overcome this problem.

By overlapping various execution stages, we've substantially reduced the number of steps, and consequently the number of clock cycles, that these instructions need to complete execution. This is a major key to improving CPU performance without cranking up the chip's clock speed. However, there's only so much to be gained from this approach alone, because instruction execution is still serialized. Starting with the next section, we'll see how to overlap the execution of adjacent instructions in order to save additional cycles.

9.5.1 *Functional Units*

As you've seen in the add instruction, the steps for adding two values and then storing their sum can't be done concurrently, because you can't store the sum until after you've computed it. Furthermore, there are some resources that the CPU can't share between steps in an instruction. There is only one data bus, and the CPU can't fetch an instruction's opcode while it is trying to store data to memory. In addition, many of the steps that make up the execution of an instruction share functional units in the CPU.

Functional units are groups of logic that perform a common operation, such as the arithmetic logical unit and the control unit. A functional unit can do only one operation at a time; you can't do two operations concurrently that use the same functional unit. To design a CPU that executes several stages in parallel, we must arrange those stages to reduce potential conflicts, or add extra logic so that two (or more) operations can occur simultaneously by executing in different functional units.

Consider again the steps that a `mov(srcMem, destReg);` instruction might require:

1. Fetch the instruction's opcode from memory.
2. Update the EIP register to hold the address of the displacement value following the opcode.
3. Decode the instruction's opcode to see what instruction it specifies.
4. Fetch the displacement value from memory to compute the source operand's effective address.
5. Update the EIP register to hold the address of the byte beyond the displacement value.
6. Compute the effective address of the source operand.
7. Fetch the value of the source operand.
8. Store the fetched value into the destination register.

The first operation uses the value of the EIP register, so we can't overlap it with the subsequent step, which adjusts the value in EIP. In addition, the first operation uses the bus to fetch the instruction opcode from memory, and because every step that follows this one depends upon this opcode, it's unlikely that we'll be able to overlap it with any other.

The second and third operations don't share any functional units, and the third operation doesn't depend upon the value of the EIP register, which is modified in the second step. Therefore, we can modify the control unit so that it combines these steps, adjusting the EIP register at the same time that it decodes the instruction. This will shave one cycle off the execution of the `mov` instruction.

The third and fourth steps, which decode the instruction's opcode and fetch the displacement value, don't look like they can be done in parallel, because you must decode the instruction's opcode to determine whether the CPU needs to fetch a displacement operand from memory. However, we can design the CPU to fetch the displacement anyway so that it's available if we need it.

Of course, there's no way to overlap the execution of steps 7 and 8 because the CPU must fetch the value before storing it away.

By combining all the steps that are possible, we might obtain the following sequence for a `mov` instruction:

1. Fetch the instruction's opcode from memory.
2. Decode the instruction's opcode to see what instruction it specifies, *and* update the EIP register to hold the address of the displacement value following the opcode.
3. Fetch the displacement value from memory to compute the source operand's effective address, *and* update the EIP register to hold the address of the byte beyond the displacement value.
4. Compute the effective address of the source operand.
5. Fetch the value of the source operand from memory.
6. Store the fetched value into the destination register.

By adding a small amount of logic to the CPU, we've shaved one or two cycles off the execution of the `mov` instruction. This simple optimization works with most of the other instructions as well.

Now consider the `loop` instruction, which has several steps that use the ALU. If the CPU has only a single ALU, it must execute these steps sequentially. However, if the CPU has multiple ALUs (that is, multiple functional units), it can execute some of these steps in parallel. For example, the CPU could decrement the value in the ECX register (using the ALU) at the same time it updates the EIP value. Note that the `loop` instruction also uses the ALU to compare the decremented ECX value against 0 (to determine if it should branch). However, there's a data dependency between incrementing ECX and comparing it with 0; therefore, the CPU can't perform both of these operations at the same time.

9.5.2 The Prefetch Queue

Now that we've looked at some simple optimization techniques, consider what happens when the `mov` instruction executes on a CPU with a 32-bit data bus. If the `mov` instruction fetches an 8-bit displacement value from memory, the CPU may wind up fetching an additional 3 bytes along with the displacement value (the 32-bit data bus lets us fetch 4 bytes in a single bus cycle). The second byte on the data bus is actually the opcode of the next instruction. If we could save this opcode until the execution of the next instruction, we could shave a cycle off its execution time because it wouldn't have to fetch the same opcode byte again.

9.5.2.1 Using Unused Bus Cycles

There are still more improvements we can make. While the `mov` instruction is executing, the CPU isn't accessing memory on every clock cycle. For example, while data is being stored into the destination register, the bus is idle. When the bus is idle, we can prefetch and save the instruction opcode and operands of the next instruction.

The hardware that does this is the *prefetch queue*. Figure 9-4 shows the internal organization of a CPU with a prefetch queue.

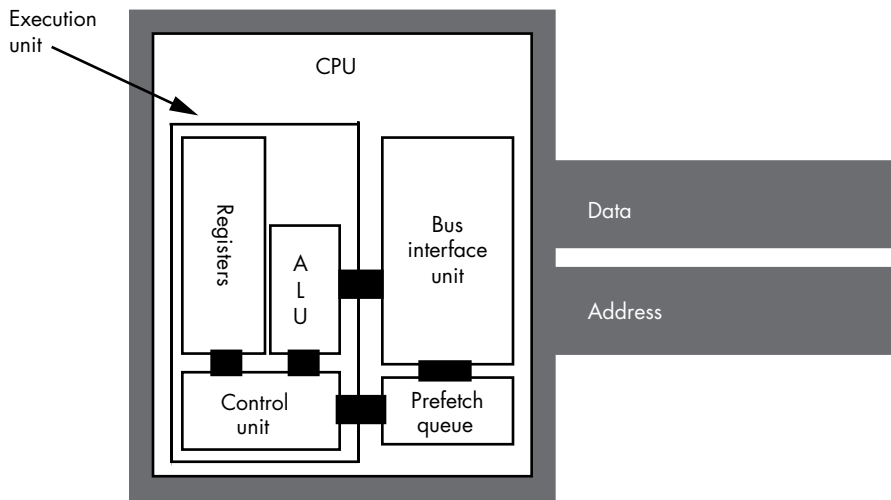


Figure 9-4: CPU design with a prefetch queue

The *bus interface unit (BIU)*, as its name implies, controls access to the address and data buses. The BIU acts as a “traffic cop” and handles simultaneous requests for bus access by different modules, such as the execution unit and the prefetch queue. Whenever some component inside the CPU wishes to access main memory, it sends this request to the BIU.

Whenever the execution unit is not using the BIU, the BIU can fetch additional bytes from the memory that holds the machine instructions and store them in the prefetch queue. Then, whenever the CPU needs an instruction opcode or operand value, it grabs *the next available byte* from

the prefetch queue. Because the BIU grabs multiple bytes at a time from memory, and because, per clock cycle, the CPU generally consumes fewer bytes from the prefetch queue than are available, instructions will normally be sitting in the prefetch queue for the CPU's use.

However, there's no guarantee that all instructions and operands will be sitting in the prefetch queue when we need them. For example, consider the 80x86 `jnz Label`; instruction. If the 2-byte form of the instruction appears at locations 400 and 401 in memory, the prefetch queue may contain the bytes at addresses 402, 403, 404, 405, 406, 407, and so on. If `jnz` transfers control to `Label` at target address 480, the bytes at addresses 402, 403, 404, and so on, won't be of any use to the CPU. The system will have to pause for a moment to fetch the data at address 480 before it can go on. Most of the time, the CPU fetches sequential values from memory, though, so having the data in the prefetch queue saves time.

9.5.2.2 Overlapping Instructions

Another improvement we can make is to overlap the processes of decoding the next instruction's opcode and executing the last step of the previous instruction. After the CPU processes the operand, the next available byte in the prefetch queue is an opcode, which the CPU can decode because the instruction decoder is idle while the CPU executes the steps of the current instruction. Of course, if the current instruction modifies the EIP register, the time the CPU spends on the decoding operation goes to waste; however, because it occurs in parallel with other operations of the current instruction, this decoding doesn't slow down the system (though it does require extra circuitry).

9.5.2.3 Summarizing Background Prefetch Events

Our instruction execution sequence now assumes that the following CPU prefetch events are occurring (concurrently) in the background:

1. If the prefetch queue is not full (generally it can hold between 8 and 32 bytes, depending on the processor) and the BIU is idle on the current clock cycle, fetch the next double word located at the address found in the EIP register at the beginning of the clock cycle.
2. If the instruction decoder is idle and the current instruction does not require an instruction operand, the CPU should begin decoding the opcode at the front of the prefetch queue. If the current instruction requires an instruction operand, then the CPU begins decoding the byte just beyond that operand in the prefetch queue.

Now let's reconsider our `mov(srcreg, destreg)`; instruction. Because we've added the prefetch queue and the BIU, we can overlap the fetch and decode stages of this instruction with specific stages of the previous instruction to get the following steps:

1. Fetch and decode the instruction; this is overlapped with the previous instruction.

2. Fetch the source register and update the EIP register with the address of the next instruction.
3. Store the fetched value into the destination register.

The instruction execution timings in this example assume that the opcode is present in the prefetch queue and that the CPU has already decoded it. If either is not true, additional cycles will be necessary to fetch the opcode from memory and decode the instruction.

9.5.3 Conditions That Hinder the Performance of the Prefetch Queue

When they transfer control to the target location, jump and conditional jump instructions are slower than other instructions, because the CPU can't overlap the processes of fetching and decoding the opcode for the next instruction with the process of executing a jump instruction that transfers control. It may take several cycles after the execution of a jump instruction for the prefetch queue to reload.

NOTE

If you want to write fast code, avoid jumping around in your program as much as possible.

Conditional jump instructions invalidate the prefetch queue only if they actually transfer control to the target location. If the jump condition is false, execution continues with the next instruction and the values in the prefetch queue remain valid. Therefore, while writing the program, if you can determine which jump condition occurs most frequently, you should arrange your program so that the most common condition causes the program to continue with the next instruction rather than jump to a separate location.

In addition, instruction size (in bytes) can affect the performance of the prefetch queue. The larger the instruction, the faster the CPU will empty the prefetch queue. Instructions involving constants and memory operands tend to be the largest. If you execute a sequence of these instructions in a row, the CPU may end up having to wait because it is removing instructions from the prefetch queue faster than the BIU is copying data to the prefetch queue. So, whenever possible, try to use shorter instructions.

Finally, prefetch queues work best when you have a wide data bus. The 16-bit 8086 processor runs much faster than the 8-bit 8088 because it can keep the prefetch queue full with fewer bus accesses. Don't forget, the CPU needs to use the bus for other purposes. Instructions that access memory compete with the prefetch queue for access to the bus. If you have a sequence of instructions that all access memory, the prefetch queue may quickly empty, and once that happens, the CPU must wait for the BIU to fetch new opcodes from memory before it can continue executing instructions.

9.5.4 Pipelining: Overlapping the Execution of Multiple Instructions

Executing instructions in parallel using a BIU and an execution unit is a special case of pipelining. Most modern processors incorporate pipelining

to improve performance. With just a few exceptions, pipelining allows us to execute one instruction per clock cycle.

The advantage of the prefetch queue is that it lets the CPU overlap the processes of fetching and decoding the instruction opcode with the execution of other instructions. Assuming you're willing to add hardware, you can execute almost all operations in parallel. That is the idea behind pipelining.

Pipelined operation improves an application's average performance by executing several instructions concurrently. However, as you saw with the prefetch queue, certain instructions (and combinations thereof) fare better than others in a pipelined system. By understanding how pipelined operation works, you can organize your applications to run faster.

9.5.4.1 A Typical Pipeline

Consider the steps necessary to do a generic operation, with each step taking one clock cycle:

1. Fetch the instruction's opcode from memory.
2. Decode the opcode *and*, if required, prefetch a displacement operand, a constant operand, or both.
3. If required, compute the effective address for a memory operand (for example, `[ebx+disp]`).
4. If required, fetch the value of any memory operand and/or register.
5. Compute the result.
6. Store the result into the destination register.

Assuming you're willing to pay for some extra silicon, you can build a little *miniprocessor* to handle each step. The organization would look something like Figure 9-5.

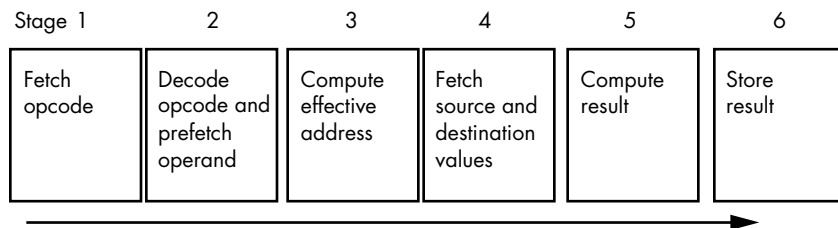


Figure 9-5: A pipelined implementation of instruction execution

In stage 4, the CPU fetches both the source and destination operands. You can set this up by putting multiple data paths inside the CPU (such as from the registers to the ALU) and ensuring that no two operands ever compete for simultaneous use of the data bus (that is, there are no memory-to-memory operations).

If you design a separate piece of hardware for each stage in the pipeline in Figure 9-5, almost all of them can take place in parallel. Of course, you can't fetch and decode the opcode for more than one instruction at the same time, but you can fetch the opcode of the next instruction while decoding the current instruction's opcode. If you have an n -stage pipeline, you will usually have n instructions executing concurrently. Figure 9-6 shows pipelining in operation. T1, T2, T3, and so on, represent consecutive "ticks" (time = 1, time = 2, and so on) of the system clock.

T1	T2	T3	T4	T5	T6	T7	T8	T9...
Opcode	Decode	Address	Values	Compute	Store	Instruction #1		
	Opcode	Decode	Address	Values	Compute	Store	Instruction #2	
		Opcode	Decode	Address	Values	Compute	Store	Instruction #3
			Opcode	Decode	Address	Values	Compute	Store

Figure 9-6: Instruction execution in a pipeline

At time $T = T1$, the CPU fetches the opcode byte for the first instruction. At $T = T2$, the CPU begins decoding the opcode for the first instruction, and, in parallel, it fetches a block of bytes from the prefetch queue in the event that the first instruction has an operand. Also in parallel with the decoding of the first instruction, the CPU instructs the BIU to fetch the opcode of the second instruction because the first instruction no longer needs that circuitry.

Note that there is a minor conflict here. The CPU is attempting to fetch the next byte from the prefetch queue for use as an operand; at the same time, it is fetching operand data from the prefetch queue for use as an opcode. How can it do both at once? You'll see the solution shortly.

At time $T = T3$, the CPU computes the address of any memory operand if the first instruction accesses memory. If the first instruction doesn't access memory, the CPU does nothing. During T3, the CPU also decodes the opcode of the second instruction and fetches any operands in the second instruction. Finally, the CPU also fetches the opcode for the third instruction. With each advancing tick of the clock, another execution stage of each instruction in the pipeline completes, and the CPU fetches the opcode of yet another instruction from memory.

This process continues until, at $T = T6$, the CPU completes the execution of the first instruction, computes the result for the second, and fetches the opcode for the sixth instruction in the pipeline. The important thing to note is that after $T = T5$, the CPU completes an instruction on every clock cycle. Once the CPU fills the pipeline, it completes one instruction on each cycle. This is true even if there are complex addressing modes to be computed, memory operands to fetch, or other operations that consume cycles on a non-pipelined processor. All you need to do is add more stages to the pipeline, and you can still effectively process each instruction in one clock cycle.

Now back to the small conflict in the pipeline organization I mentioned earlier. At $T = T_2$, for example, the CPU attempts to prefetch a block of bytes containing any operands of the first instruction, and at the same time it fetches the opcode of the second instruction. Until the CPU decodes the first instruction, it doesn't know how many operands the instruction requires or their length. Moreover, until it determines that information, the CPU doesn't know what byte to fetch as the opcode of the second instruction. So how can the pipeline fetch the opcode of the next instruction in parallel with any address operands of the current instruction?

One solution is to disallow this simultaneous operation in order to avoid the potential data hazard. If an instruction has an address or constant operand, we can simply delay the start of the next instruction. Unfortunately, many instructions have these additional operands, so this approach will substantially hinder the CPU's execution speed.

The second solution is to throw a lot more hardware at the problem. Operand and constant sizes usually come in 1-, 2-, and 4-byte lengths. Therefore, if we actually fetch the bytes in memory that are located at offsets 1, 3, and 5 bytes beyond the current opcode we are decoding, one of them will probably contain the opcode of the next instruction. Once we are through decoding the current instruction, we know how many bytes it consumes, and, therefore, we know the offset of the next opcode. We can use a simple data selector circuit to choose which of the three candidate opcode bytes we want to use.

In practice, we actually have to select the next opcode byte from more than three candidates because 80x86 instructions come in many different lengths. For example, a `mov` instruction that copies a 32-bit constant to a memory location can be 10 or more bytes long. Moreover, instructions vary in length from 1 to 15 bytes. And some opcodes on the 80x86 are longer than 1 byte, so the CPU may have to fetch multiple bytes in order to properly decode the current instruction. However, by throwing more hardware at the problem, we can decode the current opcode at the same time we're fetching the next.

9.5.4.2 Stalls in a Pipeline

Unfortunately, the scenario presented in the previous section is a little too simplistic. There are two problems that our simple pipeline ignores: competition between instructions for access to the bus (known as *bus contention*), and nonsequential instruction execution. Both problems may increase the average execution time of the instructions in the pipeline. By understanding how the pipeline works, you can write your software to avoid these pitfalls and improve the performance of your applications.

Bus contention can occur whenever an instruction needs to access an item in memory. For example, if a `mov(reg, mem);` instruction needs to store data in memory and a `mov(mem, reg);` instruction needs to fetch data from memory, contention for the address and data bus may develop because the CPU will be trying to do both operations simultaneously.

One simplistic way to handle bus contention is through a *pipeline stall*. The CPU, when faced with contention for the bus, gives priority to the instruction farthest along in the pipeline. This stalls the later instruction in the pipeline, and it takes two cycles to execute that instruction (see Figure 9-7).

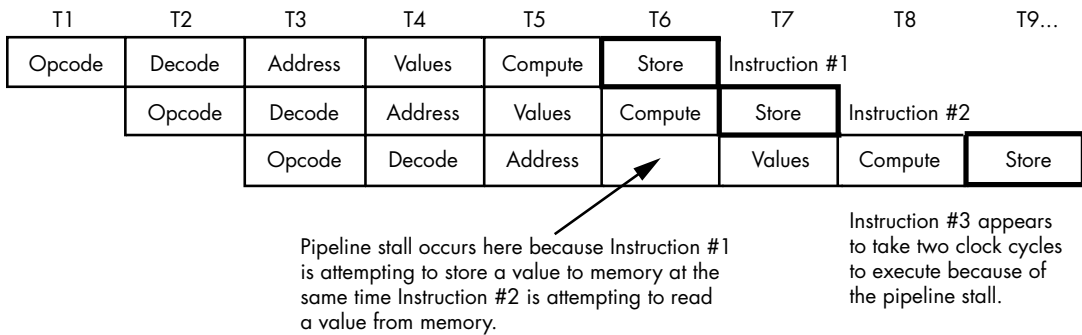


Figure 9-7: A pipeline stall

There are many other cases of bus contention. For example, fetching operands for an instruction requires access to the prefetch queue at the same time that the CPU needs to access it to fetch the opcode of the next instruction. Given the simple pipelining scheme that we've outlined so far, it's unlikely that most instructions would execute at one clock (cycle) per instruction (CPI).

As another example of a pipeline stall, consider what happens when an instruction *modifies* the value in the EIP register. For example, the `jnz label;` instruction might change the value in the EIP register if it transfers control to its target label, which implies that the next set of instructions to be executed does not immediately follow the `jnz` instruction. By the time the instruction `jnz label;` completes execution (assuming the zero flag is clear so that the branch is taken), we've already started five other instructions and we're only one clock cycle away from completing the first of these. The CPU must not execute those instructions, or it will compute improper results.

The only reasonable solution is to *flush* the entire pipeline and begin fetching opcodes anew. However, doing so causes a severe execution time penalty. It will take the length of the pipeline (six cycles in our example) before the next instruction completes execution. The longer the pipeline is, the more you can accomplish per cycle in the system, but the slower a program will run if it jumps around quite a bit. Unfortunately, you can't control the number of stages in the pipeline,⁴ but you *can* control the number of transfer instructions in your programs, so it's best to keep these to a minimum in a pipelined system.

4. Note, by the way, that the number of stages in an instruction pipeline varies among CPUs.

9.5.5 Instruction Caches: Providing Multiple Paths to Memory

System designers can resolve many problems with bus contention through the intelligent use of the prefetch queue and the cache memory subsystem. As you've seen, they can design the prefetch queue to buffer data from the instruction stream. However, they can also use a separate *instruction cache* (apart from the data cache) to hold machine instructions. As a programmer, you have no control over how your CPU's instruction cache is organized, but knowing how it operates might prompt you to use certain instruction sequences that would otherwise create stalls.

Suppose the CPU has two separate memory spaces, one for instructions and one for data, each with its own bus. This is called the *Harvard architecture* because the first such machine was built at Harvard University. On a Harvard machine, there's no contention for the bus; the BIU can continue to fetch opcodes on the instruction bus while accessing memory on the data/memory bus (see Figure 9-8).

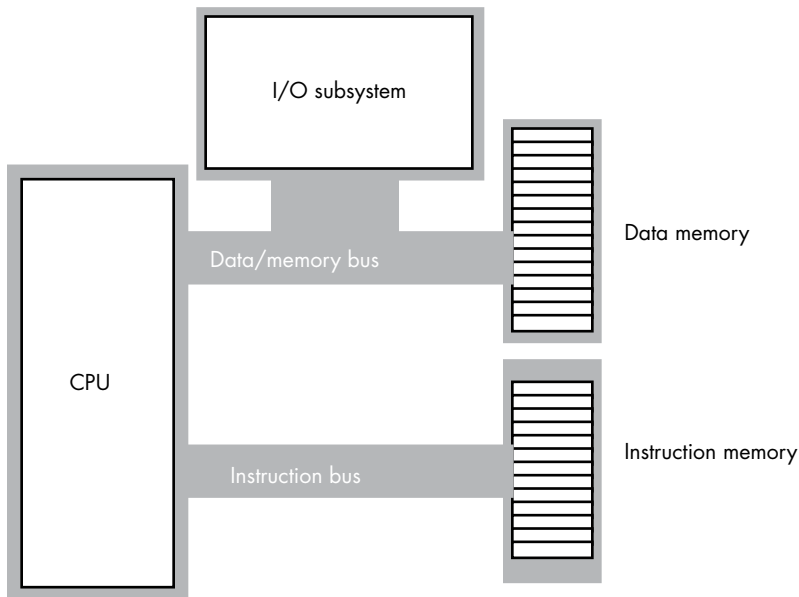


Figure 9-8: A typical Harvard machine

In the real world, there are very few true Harvard machines. The extra pins needed on the processor to support two physically separate buses increase the cost of the processor and introduce many other engineering problems. However, microprocessor designers have discovered that they can obtain many of the benefits of the Harvard architecture with few of its disadvantages by using separate on-chip caches for data and instructions. Advanced CPUs use an internal Harvard architecture and an external von Neumann architecture. Figure 9-9 shows the structure of the 80x86 with separate data and instruction caches.

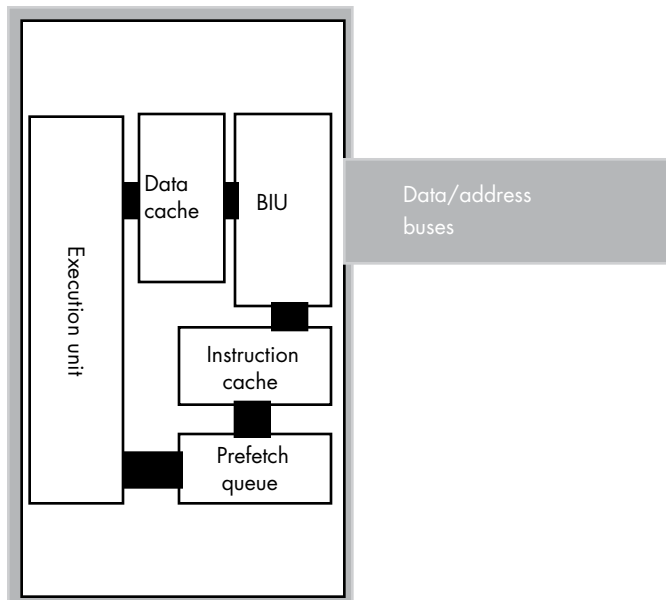


Figure 9-9: Using separate code and data caches

Each path between the sections inside the CPU represents an independent bus, and data can flow on all paths concurrently. This means that the prefetch queue can pull instruction opcodes from the instruction cache while the execution unit is writing data to the data cache. However, it's not always possible, even with a cache, to avoid bus contention. In the arrangement with two separate caches, the BIU still has to use the data/address bus to fetch opcodes from memory whenever they are not located in the instruction cache. Likewise, the data cache still has to buffer data from memory on occasion.

Although you can't control the presence, size, or type of cache on a CPU, you must be aware of how the cache operates in order to write the best programs. On-chip, level-one (L1) instruction caches are generally quite small (between 4KB and 64KB on typical CPUs) compared to the size of main memory. Therefore, the shorter your instructions, the more of them will fit in the cache (tired of "shorter instructions" yet?). The more instructions you have in the cache, the less often bus contention will occur. Likewise, using registers to hold temporary results places less strain on the data cache, so it doesn't need to flush data to memory or retrieve data from memory quite so often.

9.5.6 Pipeline Hazards

There is another problem with using a pipeline: hazards. There are two types of hazards: control hazards and data hazards. We've actually discussed control hazards already, although we didn't refer to them by name.

A control hazard occurs whenever the CPU branches to some new location in memory and consequently has to flush from the pipeline the instructions that are in various stages of execution. A data hazard occurs when two instructions attempt to access the same memory location out of sequence.

Let's take a look at data hazards using the execution profile for the following instruction sequence:

```
mov( SomeVar, ebx );
mov( [ebx], eax );
```

When these two instructions execute, the pipeline will look something like Figure 9-10.

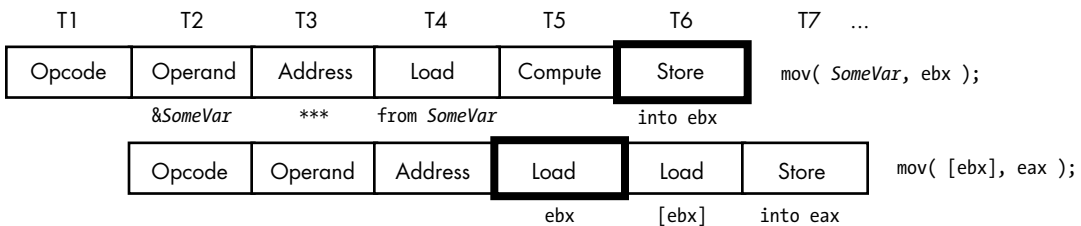


Figure 9-10: A data hazard

These two instructions attempt to fetch the 32-bit value whose address is held in the *SomeVar* pointer variable. *However, this sequence of instructions won't work properly!* The second instruction accesses the value in EBX before the first instruction copies the address of memory location *SomeVar* into EBX (T5 and T6 in Figure 9-10).

CISC processors, like the 80x86, handle hazards automatically. (Some RISC chips do not, and if you tried this sequence on certain RISC chips, you would store an incorrect value in EAX.) In order to handle the data hazard in this example, CISC processors stall the pipeline to synchronize the two instructions. The actual execution would look something like Figure 9-11.

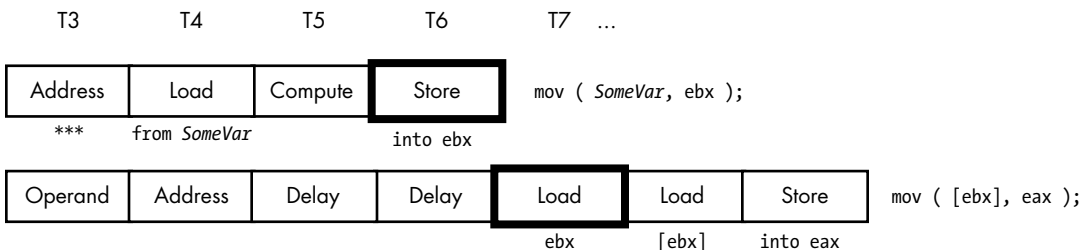


Figure 9-11: How a CISC CPU handles a data hazard

By delaying the second instruction by two clock cycles, the CPU guarantees that the load instruction will load EAX with the value at the proper address. Unfortunately, the `mov([ebx], eax);` instruction now executes in three clock cycles rather than one. However, requiring two extra clock cycles is better than producing incorrect results.

Fortunately, you (or your compiler) can reduce the impact that hazards have on program execution speed within your software. A data hazard occurs when the source operand of one instruction was a destination operand of a previous instruction. There's nothing wrong with loading EBX from *SomeVar* and then loading EAX from [EBX] (that is, the double-word memory location pointed at by EBX), *as long as they don't occur one right after the other*. Suppose the code sequence had been:

```
mov( 2000, ecx );  
mov( SomeVar, ebx );  
mov( [ebx], eax );
```

We could reduce the effect of the hazard in this code sequence by simply rearranging the instructions, as follows:

```
mov( SomeVar, ebx );  
mov( 2000, ecx );  
mov( [ebx], eax );
```

Now the `mov([ebx], eax);` instruction requires only one additional clock cycle. By inserting yet another instruction between the `mov(SomeVar, ebx);` and the `mov([ebx], eax);` instructions, you can eliminate the effects of the hazard altogether (of course, the inserted instruction must not modify the values in the EAX and EBX registers).

On a pipelined processor, the order of instructions in a program may dramatically affect the program's performance. If you're writing assembly code, always look for possible hazards and eliminate them wherever possible by rearranging your instruction sequences. If you're using a compiler, choose one that properly handles instruction ordering.

9.5.7 Superscalar Operation: Executing Instructions in Parallel

With the pipelined architecture shown so far, we could achieve, at best, execution times of one CPI. Is it possible to execute instructions faster than this? At first you might think, "Of course not—we can do at most one operation per clock cycle, so there's no way we can execute more than one instruction per clock cycle." Keep in mind, however, that a single instruction is *not* a single operation. In the examples presented earlier, each instruction took between six and eight operations to complete. By adding seven or eight separate units to the CPU, we could effectively execute these eight operations in one clock cycle, yielding one CPI. If we add more hardware and execute, say, 16 operations at once, can we achieve 0.5 CPI? The answer is a qualified yes. A CPU that includes this additional hardware is a *superscalar* CPU, and it can execute more than one instruction during a single clock cycle. The 80x86 family began supporting superscalar execution with the introduction of the Pentium processor.

A superscalar CPU has several execution units (see Figure 9-12). If it encounters in the prefetch queue two or more instructions that it can execute independently, it will do so.

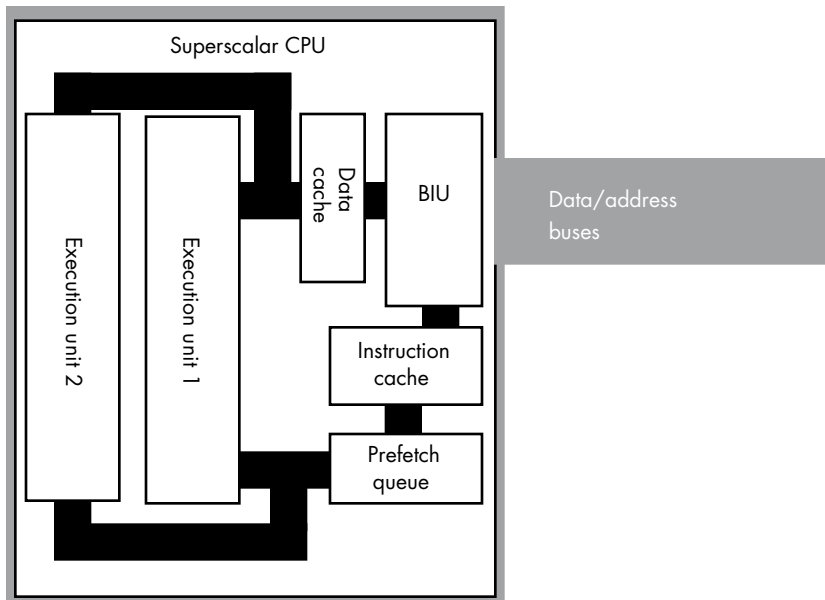


Figure 9-12: A CPU that supports superscalar operation

There are a couple of advantages to going superscalar. Suppose you have the following instructions in the instruction stream:

```
mov( 1000, eax );  
mov( 2000, ebx );
```

If there are no other problems or hazards in the surrounding code, and all 6 bytes for these two instructions are currently in the prefetch queue, there's no reason why the CPU can't fetch and execute both instructions in parallel. All it takes is extra silicon on the CPU chip to implement two execution units.

Besides speeding up independent instructions, a superscalar CPU can also speed up program sequences that have hazards. One limitation of normal CPUs is that once a hazard occurs, the offending instruction will completely stall the pipeline. Every instruction that follows the stalled instruction will also have to wait for the CPU to synchronize the execution of the offending instructions. With a superscalar CPU, however, instructions following the hazard may continue execution through the pipeline as long as they don't have hazards of their own. This alleviates (though it does not eliminate) some of the need for careful instruction scheduling.

The way you write software for a superscalar CPU can dramatically affect its performance. First and foremost is that rule you're probably sick of by now: *use short instructions*. The shorter your instructions, the more instructions the CPU can fetch in a single operation and, therefore, the more likely the CPU will execute faster than one CPI. Most superscalar CPUs do not completely duplicate the execution unit. There might be multiple ALUs, floating-point units, and so on, which means that certain

instruction sequences can execute very quickly, while others won't. You have to study the exact composition of your CPU to decide which instruction sequences produce the best performance.

9.5.8 Out-of-Order Execution

In a standard superscalar CPU, it is the programmer's (or compiler's) responsibility to arrange the instructions to avoid hazards and pipeline stalls. Fancier CPUs can actually remove some of this burden and improve performance by automatically rescheduling instructions while the program executes. To understand how this is possible, consider the following instruction sequence:

```
mov( SomeVar, ebx );  
mov( [ebx], eax );  
mov( 2000, ecx );
```

There's a data hazard between the first and second instructions. The second instruction must delay until the first instruction completes execution. This introduces a pipeline stall and increases the running time of the program. Typically, the stall affects every instruction that follows. However, the third instruction's execution does not depend on the result from either of the first two instructions. Therefore, there's no reason to stall the execution of the `mov(2000, ecx);` instruction. It can continue executing while the second instruction waits for the first to complete. This technique is called *out-of-order execution* because the CPU can execute instructions prior to the completion of instructions appearing previously in the code stream.

Keep in mind that the CPU can execute instructions out of sequence only if doing so produces exactly the same results as sequential execution. While there are many little technical issues that make this feature more difficult than it seems, with enough engineering effort you can implement it.

9.5.9 Register Renaming

One problem that hampers the effectiveness of superscalar operation on the 80x86 CPU is its limited number of general-purpose registers. Suppose, for example, that the CPU had four different pipelines and, therefore, was capable of executing four instructions simultaneously. Presuming no conflicts existed among these instructions and they could all execute simultaneously, it would still be very difficult to actually achieve four instructions per clock cycle because most instructions operate on two register operands. For four instructions to execute concurrently, you'd need eight different registers: four destination registers and four source registers (none of the destination registers could double as source registers of other instructions). CPUs that have lots of registers can handle this task quite easily, but the limited register set of the 80x86 makes this difficult. Fortunately, there's a trick to alleviate part of the problem: *register renaming*.

Register renaming is a sneaky way to give a CPU more registers than it actually has. Programmers won't have direct access to these extra registers,

but the CPU can use them to prevent hazards in certain cases. For example, consider the following short instruction sequence:

```
mov( 0, eax );  
mov( eax, i );  
mov( 50, eax );  
mov( eax, j );
```

There's a data hazard between the first and second instructions as well as between the third and fourth instructions. Out-of-order execution in a superscalar CPU would normally allow the first and third instructions to execute concurrently, and then the second and fourth instructions could execute concurrently. However, there's also a data hazard between the first and third instructions because they use the same register. The programmer could have easily solved this problem by using a different register (say, EBX) for the third and fourth instructions. However, let's assume that the programmer was unable to do this because all the other registers were holding important values. Is this sequence doomed to executing in four cycles on a superscalar CPU that should require only two?

One advanced trick a CPU can employ is to create a bank of registers for each of the general-purpose registers on the CPU. That is, rather than having a single EAX register, the CPU could support an array of EAX registers; let's call these registers EAX[0], EAX[1], EAX[2], and so on. Similarly, you could have an array of each of the other registers: EBX[0] through EBX[*n*], ECX[0] through ECX[*n*], and so on. The instruction set doesn't permit the programmer to select one of these specific register array elements for a given instruction, but the CPU can automatically choose among them if doing so wouldn't change the overall computation and could speed up program execution. This is known as *register renaming*. For example, consider the following sequence (with register array elements automatically chosen by the CPU):

```
mov( 0, eax[0] );  
mov( eax[0], i );  
mov( 50, eax[1] );  
mov( eax[1], j );
```

Because EAX[0] and EAX[1] are different registers, the CPU can execute the first and third instructions concurrently. Likewise, the CPU can execute the second and fourth instructions concurrently.

Although this is a simple example, and different CPUs implement register renaming in many different ways, you can see how the CPU can use this technique to improve performance.

9.5.10 VLIW Architecture

Superscalar operation attempts to schedule, in hardware, the execution of multiple instructions simultaneously. Another technique, which Intel is using in its IA-64 architecture, involves *very long instruction words* (VLIW).

In a VLIW computer system, the CPU fetches a large block of bytes (41 bits in the case of the IA-64 Itanium CPU) and decodes and executes it all at once. This block of bytes usually contains two or more instructions (three in the case of the IA-64). VLIW computing requires the programmer or compiler to properly schedule the instructions in each block so that there are no hazards or other conflicts, but if all goes well, the CPU can execute three or more instructions per clock cycle.

9.5.11 Parallel Processing

Most techniques for improving CPU performance via architectural advances involve the parallel execution of instructions. If programmers are aware of the underlying architecture, they can write code that runs faster, but these architectural advances often improve performance significantly even if programmers do not write special code to take advantage of them.

The only problem with ignoring the underlying architecture is that there's only so much the hardware can do to parallelize a program that requires sequential execution for proper operation. To truly produce a parallel program, the programmer must specifically write parallel code, though, of course, this requires architectural support from the CPU. This section and the next touch on the types of support a CPU can provide.

Common CPUs use what's known as the *single instruction, single data* (SISD) model. This means that the CPU executes one instruction at a time, and that instruction operates on a single piece of data.⁵ Two common parallel models are the *single instruction, multiple data* (SIMD) and *multiple instruction, multiple data* (MIMD) models. Many modern CPUs, including the 80x86, include limited support for these parallel-execution models, providing a hybrid SISD/SIMD/MIMD architecture.

In the SIMD model, the CPU executes a single instruction stream, just like the pure SISD model, but operates on multiple pieces of data concurrently. For example, consider the 80x86 `add` instruction. This is a SISD instruction that operates on (that is, produces) a single piece of data. True, the instruction fetches values from two source operands, but the end result is that the `add` instruction stores a sum into only a single destination operand. An SIMD version of `add`, on the other hand, would compute several sums simultaneously. The 80x86 MMX and SIMD instruction extensions, the ARM's Neon instructions, and the PowerPC's AltiVec instructions, operate in exactly this fashion. With the `paddb` MMX instruction, for example, you can add up to eight separate pairs of values with the execution of a single instruction. Here's an example of this instruction:

```
paddb( mm0, mm1 );
```

5. We'll ignore the parallelism provided by pipelining and superscalar operation in this discussion.

Although this instruction appears to have only two operands (like a typical SISD add instruction on the 80x86), the MMX registers (MM0 and MM1) actually hold eight independent byte values (the MMX registers are 64 bits wide but are treated as eight 8-bit values).

Unless you have an algorithm that can take advantage of SIMD instructions, they're not that useful. Fortunately, high-speed 3D graphics and multimedia applications benefit greatly from these SIMD (and MMX) instructions, so their inclusion in the 80x86 CPU offers a huge performance boost for these important applications.

The MIMD model uses multiple instructions, operating on multiple pieces of data (usually with one instruction per data object, though one of these instructions could also operate on multiple data items). These multiple instructions execute independently of one another, so it's very rare that a single program (or, more specifically, a single thread of execution) would use the MIMD model. However, if you have a multiprogramming environment with multiple programs attempting to execute concurrently, the MIMD model does allow each of those programs to execute its own code stream simultaneously. This type of parallel system is called a *multiprocessor system*.

9.5.12 Multiprocessing

Pipelining, superscalar operation, out-of-order execution, and VLIW designs are all techniques that CPU designers use in order to execute several operations in parallel. These techniques support *fine-grained parallelism* and are useful for speeding up adjacent instructions in a computer system. If adding more functional units increases parallelism, what would happen if you added another CPU to the system? This approach, known as *multiprocessing*, can improve system performance, though not as uniformly as other techniques.

Multiprocessing doesn't help a program's performance unless that program is specifically written for use on a multiprocessor system. If you build a system with two CPUs, those CPUs cannot trade off executing alternate instructions within a single program. It is very expensive, time-wise, to switch the execution of a program's instructions from one processor to another. Therefore, multiprocessor systems are effective only with an operating system that executes multiple processes or threads concurrently. To differentiate this type of parallelism from that afforded by pipelining and superscalar operation, we'll call this *coarse-grained parallelism*.

Adding multiple processors to a system is not as simple as wiring two or more processors to the motherboard. To understand why this is so, consider two separate programs running on separate processors in a multiprocessor system. These two processors communicate with each other by writing to a block of shared physical memory. When CPU 1 writes to this block of memory it caches the data locally and might not actually write the data to physical memory for some time. If CPU 2 attempts to simultaneously read this block of shared memory, it winds up reading the old data out of main memory (or its local cache) rather than reading the updated data that CPU 1 wrote to its local cache. This is known as the *cache-coherency* problem.

In order for these two functions to operate properly, the two CPUs must notify each other whenever they make changes to shared objects, so the other CPU can update its own locally cached copy.

Multiprocessing is an area where the RISC CPUs have a big advantage over Intel's CPUs. While Intel 80x86 systems reach a point of diminishing returns at around 32 processors, Sun SPARC and other RISC processors easily support 64-CPU systems (with more arriving, it seems, every day). This is why large databases and large web server systems tend to use expensive Unix-based RISC systems rather than 80x86 systems.

Newer versions of the Intel i-series and Xeon processors support a hybrid form of multiprocessing known as *hyperthreading*. The idea behind hyperthreading is deceptively simple—in a typical superscalar processor it's rare for an instruction sequence to utilize all the CPU's functional units on each clock cycle. Rather than allow those functional units to go unused, the CPU can run two separate threads of execution concurrently and keep all the functional units occupied. This allows a single CPU to effectively do the work of 1.5 CPUs in a typical multiprocessor system.

9.6 For More Information

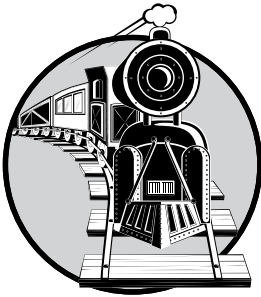
Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Elsevier, 2012.

NOTE

One subject missing from this chapter is the design of the CPU's actual instruction set. That is the subject of the next chapter.

10

INSTRUCTION SET ARCHITECTURE



This chapter discusses the implementation of a CPU's instruction set. Although the choice of a given instruction set is usually beyond a software engineer's control, understanding the decisions a hardware design engineer has to make when designing a CPU's instruction set can definitely help you write better code.

CPU instruction sets contain several tradeoffs based on assumptions that computer architects make about the way software engineers write code. If the machine instructions you choose match those assumptions, your code will probably run faster and require fewer machine resources. Conversely, if your code violates the assumptions, chances are pretty good it won't perform as well as it otherwise could.

Although studying the instruction set may seem like a task suited only to assembly language programmers, even high-level language programmers can benefit from doing so. After all, every HLL statement maps to some sequence of machine instructions, and the general concepts of instruction

set design are portable across architectures. Even if you never intend to write software using assembly language, it's important to understand how the underlying machine instructions work and how they were designed.

10.1 The Importance of Instruction Set Design

While features like caches, pipelining, and superscalar implementation can all be grafted onto a CPU long after the original design is obsolete, it's very difficult to change the instruction set once a CPU is in production and people are writing software using it. Therefore, instruction set design requires very careful consideration; the designer must get the *instruction set architecture (ISA)* correct from the start of the design cycle.

You might assume that the “kitchen sink” approach to instruction set design—in which you include every instruction you can dream up—is best. However, instruction set design is the epitome of compromise management. Why can't we have it all? Well, in the real world there are some nasty realities that prevent this:

Silicon real estate The first nasty reality is that each feature requires some number of transistors on the CPU's silicon die (chip), so CPU designers have a “silicon budget”—a finite number of transistors to work with. There simply aren't enough transistors to support putting every possible feature on a CPU. The original 8086 processor, for example, had a silicon budget of fewer than 30,000 transistors. The 1999 Pentium III processor had a budget of over 9 million transistors. The 2019 AWS Graviton2 (ARM) CPU has over 30 billion transistors.¹ These three budgets reflect the differences in semiconductor technology from 1978 to today.

Cost Although it's possible to use billions of transistors on a CPU today, the more transistors used, the more expensive the CPU. For example, at the beginning of 2018, Intel i7 processors using billions of transistors cost hundreds of dollars, whereas contemporary CPUs with 30,000 transistors cost less than a dollar.

Expandability It's very difficult to anticipate all the features people will want. For example, Intel's MMX and SIMD instruction enhancements were added to make multimedia programming more practical on the Pentium processor. Back in 1978, when Intel created the first 8086 processor, very few people could have predicted the need for these instructions. A CPU designer must allow for making extensions to the instruction set in future members of the CPU family to accommodate currently unanticipated needs.

Legacy support for old instructions This nasty reality is almost the opposite of expandability. Often, an instruction that the CPU designer feels is important now turns out to be less useful than expected. For

1. Though this is a bit of an outlier, typical desktop and server CPUs circa 2019/2020 contained 5 to 10 billion transistors.

example, the `loop` and `enter` instructions on the 80x86 CPU see very little use in modern high-performance programs. It's commonly the case that programs never use some of the instructions in a CPU adopting the kitchen sink approach. Unfortunately, once an instruction is added to the instruction set, it has to be supported in all future versions of the processor, unless few enough programs use the instruction that CPU designers are willing to let those programs break.

Complexity A CPU designer must consider the assembly programmers and compiler writers who will be using the chip. A CPU employing the kitchen sink approach might appeal to someone who's already familiar with that CPU, but no one else will want to learn an overly complex system.

These problems with the kitchen sink approach all have a common solution: design a simple instruction set for the first version of the CPU, and leave room for later expansion. This is one of the main reasons the 80x86 has proven to be so popular and long-lived. Intel started with a relatively simple CPU and figured out how to extend the instruction set over the years to accommodate new features.

10.2 Basic Instruction Design Goals

The efficiency of your programs largely depends upon the instructions that they use. Short instructions use very little memory and often execute rapidly, but they can't tackle big tasks. Larger instructions can handle more complex tasks, with a single instruction often doing the work of several short instructions, but they may consume excessive memory or require many machine cycles to execute. To enable software engineers to write the best possible code, computer architects must strike a balance between the two.

In a typical CPU, the computer encodes instructions as numeric values (operation codes, or *opcodes*) and stores them in memory. Encoding these instructions is one of the major tasks in instruction set design, requiring careful thought. Instructions must each have a unique opcode, so the CPU can differentiate them. With an n -bit number, there are 2^n different possible opcodes, so to encode m instructions requires at least $\log_2(m)$ bits. The main point to keep in mind is that the size of individual CPU instructions is dependent on the total number of instructions that the CPU supports.

Encoding opcodes is a little more involved than assigning a unique numeric value to each instruction. As the previous chapter discussed, decoding each instruction and executing the specified task requires actual circuitry. With a 7-bit opcode, we could encode 128 different instructions. To decode each of these 128 instructions requires a 7- to 128-line decoder—an expensive piece of circuitry. However, assuming that the instruction opcodes contain certain (binary) patterns, a single large decoder can often be replaced by several smaller, less expensive ones.

If an instruction set contains 128 unrelated instructions, there's little you can do other than decode the entire bit string for each instruction. However, in most architectures the instructions fall into categories. On

the 80x86 CPUs, for example, `mov(eax, ebx)`; and `mov(ecx, edx)`; have different opcodes, because they're different instructions, but they're obviously related in that they both move data from one register to another. The only difference is their source and destination operands. Thus, CPU designers could encode instructions like `mov` with a *subopcode*, and then they could encode the instruction's operands using other bit fields within the opcode.

For example, given an instruction set with only eight instructions, each with two operands, and each operand having only one of four possible values, we could encode the instructions using three packed fields containing 3, 2, and 2 bits, respectively (see Figure 10-1).

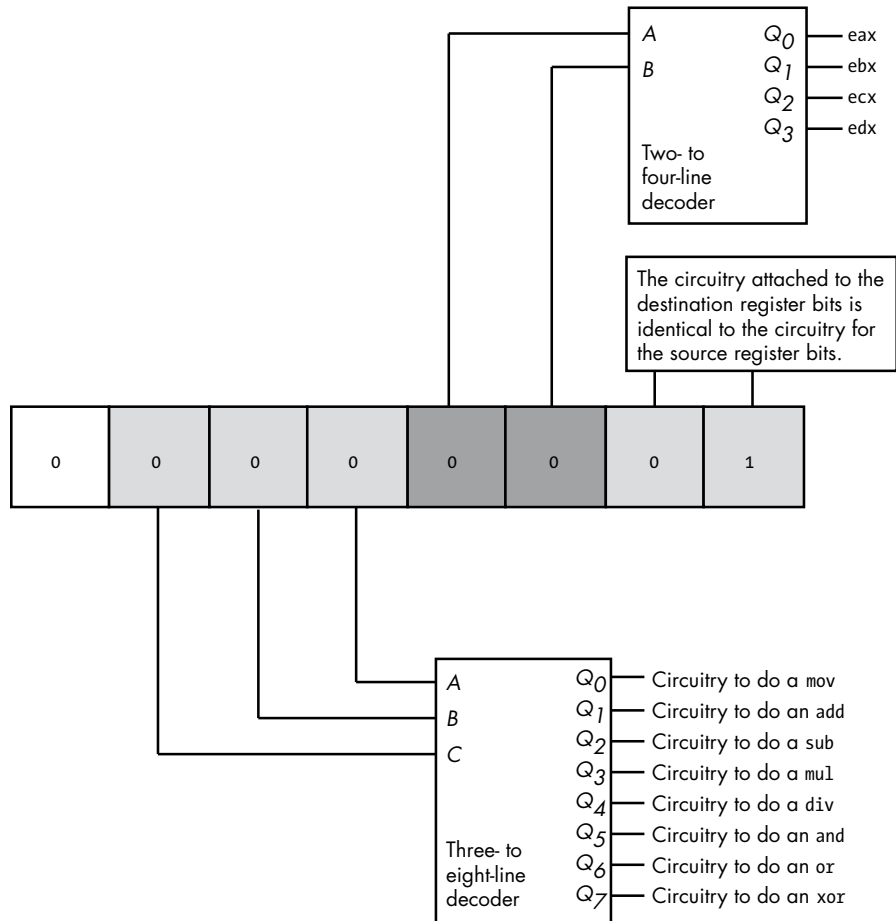


Figure 10-1: Separating an opcode into several fields to simplify decoding

This encoding needs only three simple decoders to determine what the CPU should do. While this is a basic example, it demonstrates one very important facet of instruction set design: opcodes should be easy to decode. The easiest way to simplify the opcode is to construct it using several different bit fields. The smaller these bit fields are, the easier it will be for the hardware to decode and execute the instruction.

The CPU designer's goal, then, is to assign an appropriate number of bits to the opcode's instruction field and to its operand fields. Choosing more bits for the instruction field lets the opcode encode more instructions, just as choosing more bits for the operand fields lets the opcode specify a larger number of operands (often memory locations or registers). You might think that when encoding 2^n different instructions using n bits, you'd have very little leeway in choosing the size of the instruction. It's going to take n bits to encode those 2^n instructions; you can't do it with any fewer. It *is* possible, however, to use more than n bits. This might seem wasteful, but sometimes it's advantageous. Again, picking an appropriate instruction size is one of the more important aspects of instruction set design.

10.2.1 Choosing Opcode Length

Opcode length isn't arbitrary. Assuming that a CPU is capable of reading bytes from memory, the opcode will probably have to be some multiple of 8 bits long. If the CPU is not capable of reading bytes from memory (most RISC CPUs read memory only in 32- or 64-bit chunks), the opcode will be the same size as the smallest object the CPU can read from memory at one time. Any attempt to shrink the opcode size below this limit is futile. In this chapter, we'll work with the first case: opcodes that must have a length that is a multiple of 8 bits.

Another point to consider is the size of an instruction's operands. Some CPU designers include all operands in their opcode. Other CPU designers don't count operands like immediate constants or address displacements as part of the opcode, and this is the approach we'll take.

An 8-bit opcode can encode only 256 different instructions. Even if we don't count instruction operands as part of the opcode, having only 256 different instructions is a stringent limit. Though CPUs with 8-bit opcodes exist, modern processors tend to have far more than 256 different instructions. Because opcodes must have a length that is a multiple of 8 bits, the next smallest possible opcode size is 16 bits. A 2-byte opcode can encode up to 65,536 different instructions, though the instructions will be larger.

When reducing instruction size is an important design goal, CPU designers often employ data compression theory. The first step is to analyze programs written for a typical CPU and count how many times each instruction occurs over a large number of applications. The second step is to create a list of these instructions, sorted by their frequency of use. Next, the designer assigns the 1-byte opcodes to the most frequently used instructions; 2-byte opcodes to the next most frequently used instructions; and opcodes of 3 or more bytes to the rarely used instructions. Although this scheme requires opcodes with a maximum size of 3 or more bytes, most of the actual instructions in a program will use 1- or 2-byte opcodes. The average opcode length will be somewhere between 1 and 2 bytes (let's say 1.5 bytes), and a typical program will be shorter than had all the instructions employed a 2-byte opcode (see Figure 10-2).

0	1	X	X	X	X	X	X
1	0	X	X	X	X	X	X
1	1	X	X	X	X	X	X

If the HO 2 bits of the first opcode byte are not both 0, then the whole opcode is 1 byte long, and the remaining 6 bits let us encode 64 1-byte instructions. Because there are a total of three opcode bytes of this form, we can encode up to 192 different 1-byte instructions.

0	0	1	X	X	X	X	X
---	---	---	---	---	---	---	---

X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---

If the HO 3 bits of our first opcode byte contain %001, then the opcode is 2 bytes long, and the remaining 13 bits let us encode 8,192 different instructions.

0	0	0	X	X	X	X	X
---	---	---	---	---	---	---	---

X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---

X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---

If the HO 3 bits of our first opcode byte contain all 0s, then the opcode is 3 bytes long, and the remaining 21 bits let us encode two million (2^{21}) different instructions.

Figure 10-2: Encoding instructions using a variable-length opcode

Although using variable-length instructions allows us to create smaller programs, it comes at a price. First, decoding variable-length instructions is a bit more complicated than decoding fixed-length instructions. Before decoding a particular instruction field, the CPU must first decode the instruction's size, which consumes time. This may affect the CPU's overall performance by introducing delays in the decoding step, which in turn limits the CPU's maximum clock speed (because those delays stretch out a single clock period, thus reducing the CPU's clock frequency). Variable-length instructions also make decoding multiple instructions in a pipeline difficult, because the CPU can't easily determine the instruction boundaries in the prefetch queue.

For these reasons and others, most popular RISC architectures avoid variable-length instructions. However, in this chapter, we'll study a variable-length approach, because saving memory is an admirable goal.

10.2.2 Planning for the Future

Before actually choosing the instructions to implement in a CPU, designers must plan for the future. As explained earlier, the need for new instructions will undoubtedly arise after the initial design, so it's wise to reserve some opcodes specifically for expansion purposes. Given the instruction opcode format in Figure 10-2, it might not be a bad idea to reserve one block of 64 1-byte opcodes, half (4,096) of the 2-byte opcodes, and half (1,048,576) of the 3-byte opcodes for future use. Giving up 64 of the very valuable 1-byte opcodes may seem extravagant, but history suggests that such foresight is rewarded.

10.2.3 Choosing Instructions

The next step is to choose the instructions to implement. Even if nearly half the instructions have been reserved for future expansion, that doesn't mean that all the remaining opcodes must be used to implement instructions. A designer can leave a number of these instructions unimplemented, effectively reserving them for the future as well. The right approach is not to use up the opcodes as quickly as possible, but rather to produce a consistent and complete instruction set given the design compromises. It's much easier to add an instruction later than it is to remove one, so, for the first go-round, it's generally better to go with a simpler design.

First, choose some generic instruction types. Early in the design process it's important to limit your choices to very common instructions. Other processors' instruction sets are probably the best place to look for suggestions. For example, most processors have the following:

- Data movement instructions (such as `mov`)
- Arithmetic and logical instructions (such as `add`, `sub`, `and`, `or`, `not`)
- Comparison instructions
- Conditional jump instructions (generally used after the comparison instructions)

- Input/output instructions
- Other miscellaneous instructions

The initial instruction set should comprise a reasonable number of instructions that will allow programmers to write efficient programs, without exceeding the silicon budget or violating other design constraints. This requires CPU designers to make strategic decisions based on careful research, experimentation, and simulation.

10.2.4 *Assigning Opcodes to Instructions*

After choosing the initial instructions, the CPU designer assigns opcodes to them. The first step in this process is to group the instructions according to the characteristics they share. For example, an add instruction probably supports the exact same set of operands as the sub instruction, so it makes sense to group these two instructions together. On the other hand, the not and neg instructions each generally require only a single operand. Therefore, it makes sense to put these two instructions in the same group, but one separate from the add and sub group.

Once all the instructions are grouped, the next step is to encode them. A typical encoding scheme uses some bits to select the group, some to select a particular instruction from that group, and some to encode the operand types (such as registers, memory locations, and constants). The number of bits needed to encode all this information can have a direct impact on the instruction's size, regardless of how often the instruction is used. For example, suppose 2 bits are needed to select an instruction's group, 4 bits to select the instruction within that group, and 6 bits to specify the instruction's operand types. In this case, the instructions are not going to fit into an 8-bit opcode. On the other hand, if all we need is to push one of eight different registers onto the stack, 4 bits will be enough to specify the push instruction group, and 3 bits will be enough to specify the register.

Encoding instruction operands with a minimal amount of space is always a problem, because many instructions allow a large number of operands. For example, the generic 32-bit 80x86 mov instruction allows two operands and requires a 2-byte opcode.² However, Intel noticed that `mov(dis, eax)`; and `mov(eax, disp)`; occur frequently in programs, so it created a special 1-byte version of these instructions to reduce their size and, consequently, the size of programs that use them. Intel did not remove the 2-byte versions of these instructions, though: there are two different instructions that store EAX into memory and two different instructions that load EAX from memory. A compiler or assembler will always emit the shorter versions of each pair of instructions.

Intel made an important tradeoff with the mov instruction: it gave up an extra opcode in order to provide a shorter version of one variant of each instruction. Actually, Intel uses this trick all over the place to create

2. Actually, Intel claims it's a 1-byte opcode plus a 1-byte mod-reg-r/m byte. For our purposes, we'll treat the mod-reg-r/m byte as part of the opcode.

shorter and easier-to-decode instructions. Back in 1978, creating redundant instructions to reduce program size was a good compromise given the cost of memory. Today, however, a CPU designer would probably use those redundant opcodes for different purposes.

10.3 The Y86 Hypothetical Processor

Because of enhancements made to the 80x86 processor family over time, Intel's design goals in 1978, and the evolution of computer architecture, the encoding of 80x86 instructions is very complex and somewhat illogical. In short, the 80x86 is not a good introductory example of instruction set design. To work around this, we'll discuss instruction set design in two stages: first, we'll develop a trivial instruction set for the Y86, a hypothetical processor that is a small subset of the 80x86, and then we'll expand our discussion to the full 80x86 instruction set.

10.3.1 Y86 Limitations

The hypothetical Y86 processor is a *very* stripped-down version of the 80x86 CPUs. It supports only:

- One operand size: 16 bits. This simplification frees us from having to encode the operand size as part of the opcode (thereby reducing the total number of opcodes we'll need).
- Four 16-bit registers: AX, BX, CX, and DX. This lets us encode register operands with only 2 bits (versus the 3 bits the 80x86 family requires to encode eight registers).
- A 16-bit address bus with a maximum of 65,536 bytes of addressable memory.

These simplifications, plus a very limited instruction set, will allow us to encode all Y86 instructions using a 1-byte opcode and a 2-byte displacement/offset when applicable.

10.3.2 Y86 Instructions

Including both forms of the `mov` instruction, the Y86 CPU still provides only 18 basic instructions. Seven of these instructions have two operands, eight have one operand, and five have no operands at all. The instructions are `mov` (two forms), `add`, `sub`, `cmp`, `and`, `or`, `not`, `je`, `jne`, `jb`, `jbe`, `ja`, `jae`, `jmp`, `get`, `put`, and `halt`.

10.3.2.1 The `mov` Instruction

The `mov` instruction comes in two forms, merged into the same instruction class:

```
mov( reg/memory/constant, reg );  
mov( reg, memory );
```

In these forms, *reg* is either register *ax*, *bx*, *cx*, or *dx*; *memory* is an operand specifying a memory location; and *constant* is a numeric constant using hexadecimal notation.

10.3.2.2 Arithmetic and Logical Instructions

The arithmetic and logical instructions are as follows:

```
add( reg/memory/constant, reg );
sub( reg/memory/constant, reg );
cmp( reg/memory/constant, reg );
and( reg/memory/constant, reg );
or( reg/memory/constant, reg );

not( reg/memory );
```

The `add` instruction adds the value of the first operand to the value of the second, storing the sum in the second operand. The `sub` instruction subtracts the value of the first operand from the value of the second, storing the difference in the second operand. The `cmp` instruction compares the value of the first operand against the value of the second and saves the result of the comparison for use by the conditional jump instructions (described in the next section). The `and` and `or` instructions compute bitwise logical operations between their two operands and store the result in the second operand. The `not` instruction appears separately because it supports only a single operand. `not` is the bitwise logical operation that inverts the bits of its single memory or register operand.

10.3.2.3 Control Transfer Instructions

The *control transfer instructions* interrupt the execution of instructions stored in sequential memory locations and transfer control to instructions stored at some other point in memory. They do this either unconditionally, or conditionally based upon the result from a `cmp` instruction. These are the control transfer instructions:

```
ja  dest; // Jump if above (i.e., greater than)
jae dest; // Jump if above or equal (i.e., greater than or equal to)
jb  dest; // Jump if below (i.e., less than)
jbe dest; // Jump if below or equal (i.e., less than or equal to)
je  dest; // Jump if equal
jne dest; // Jump if not equal

jmp dest; // Unconditional jump
```

The first six instructions (`ja`, `jae`, `jb`, `jbe`, `je`, and `jne`) let you check the result of the previous `cmp` instruction—that is, the result of the comparison of that instruction’s first and second operands.³ For example, if you

3. The Y86 processor performs only *unsigned* comparisons.

compare the AX and BX registers with a `cmp(ax, bx)`; instruction and then execute the `ja` instruction, the Y86 CPU will jump to the specified destination location if AX is greater than BX. If AX is not greater than BX, control will fall through to the next instruction in the program. In contrast to the first six instructions, the `jmp` instruction unconditionally transfers control to the instruction at the destination address.

10.3.2.4 Miscellaneous Instructions

The Y86 supports three instructions that do not have any operands:

```
get;    // Read an integer value into the AX register
put;    // Display the value in the AX register
halt;   // Terminate the program
```

The `get` and `put` instructions let you read and write integer values: `get` prompts the user for a hexadecimal value and then stores that value into the AX register; `put` displays the value of the AX register in hexadecimal format. The `halt` instruction terminates program execution.

10.3.3 Operand Types and Addressing Modes on the Y86

Before assigning opcodes, we need to look at the operands these instructions support. The 18 Y86 instructions use five different operand types: registers, constants, and three memory-addressing modes (the *indirect* addressing mode, the *indexed* addressing mode, and the *direct* addressing mode). See Chapter 6 for more details on these addressing modes.

10.3.4 Encoding Y86 Instructions

Because a real CPU uses logic circuitry to decode the opcodes and act appropriately on them, it's not a good idea to arbitrarily assign opcodes to machine instructions. Instead, a typical CPU opcode uses a certain number of bits to denote the instruction class (such as `mov`, `add`, and `sub`), and a certain number of bits to encode each operand.

A typical Y86 instruction takes the form shown in Figure 10-3.

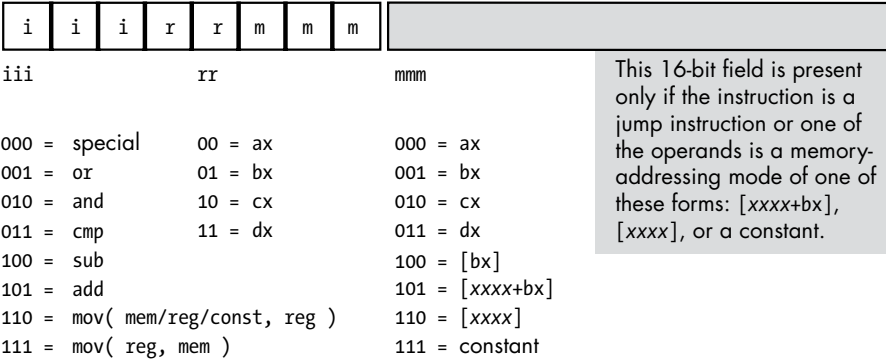


Figure 10-3: Basic Y86 instruction encoding

The basic instruction is either 1 or 3 bytes long, and its opcode consists of a single byte containing three fields. The first field, consisting of the HO 3 bits, defines the instruction, and these 3 bits provide eight possible combinations. As there are 18 different Y86 instructions, we'll have to pull some tricks to handle the remaining 10 instructions.

10.3.4.1 Eight Generic Y86 Instructions

As you can see in Figure 10-3, seven of the eight basic opcodes encode the `or`, `and`, `cmp`, `sub`, and `add` instructions, as well as both versions of the `mov` instruction. The eighth, 000, is an *expansion opcode*. This special instruction class, which we'll return to shortly, provides a mechanism that allows us to expand the set of available instructions.

To determine the full opcode for a particular instruction, you simply select the appropriate bits for the `iii`, `rr`, and `mmm` fields (identified in Figure 10-3). The `rr` field contains the destination register (except for the version of the `mov` instruction whose `iii` field is 111), and the `mmm` field encodes the source register. For example, to encode the `mov(bx, ax);` instruction you would select `iii = 110` (`mov(reg, reg);`), `rr = 00` (`ax`), and `mmm = 001` (`bx`). This produces the 1-byte instruction `%11000001`, or `$c0`.

Some Y86 instructions are larger than 1 byte. To illustrate why this is necessary, take, for example, the instruction `mov([1000], ax);`, which loads the AX register with the value stored at memory location `$1000`. The encoding for the opcode is `%11000110`, or `$c6`. However, the encoding for the `mov([2000], ax);` instruction is also `$c6`. Clearly these two instructions do different things: one loads the AX register from memory location `$1000`, while the other loads the AX register from memory location `$2000`.

In order to differentiate between instructions that encode an address using the `[xxxx]` or `[xxxx+bx]` addressing modes, or to encode a constant using the immediate addressing mode, you must append the 16-bit address or constant to the instruction's opcode. Within this 16-bit address or constant, the LO byte follows the opcode in memory and the HO byte follows the LO byte. So, the 3-byte encoding for `mov([1000], ax);` would be `$c6, $00, $10`, and the 3-byte encoding for `mov([2000], ax);` would be `$c6, $00, $20`.

10.3.4.2 The Special Expansion Opcode

The special opcode in Figure 10-3 allows the Y86 CPU to expand the set of available instructions that can be encoded using a single byte. This opcode handles several zero- and one-operand instructions, as shown in Figures 10-4 and 10-5.

Figure 10-4 shows the encodings of four one-operand instruction classes. The first 2-bit encoding for the `rr` field, `%00`, further expands the instruction set by providing a way to encode the zero-operand instructions shown in Figure 10-5. Five of these instructions are illegal instruction opcodes; the three valid opcodes are the `halt` instruction, which terminates program execution; the `get` instruction, which reads a hexadecimal value from the user and stores it in the AX register; and the `put` instruction, which outputs the value in the AX register.

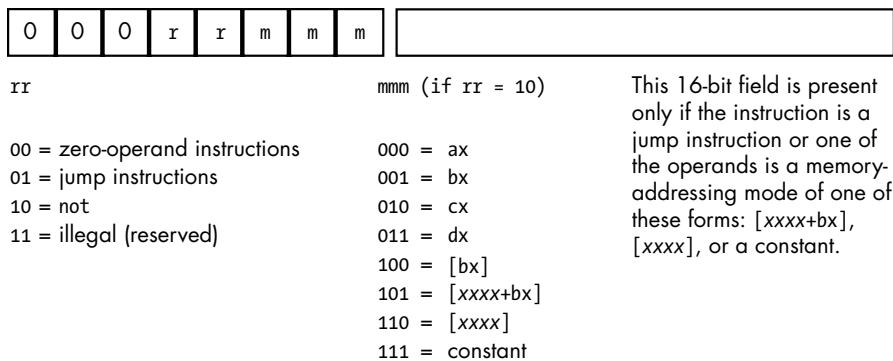


Figure 10-4: Single-operand instruction encodings (iii = %000)

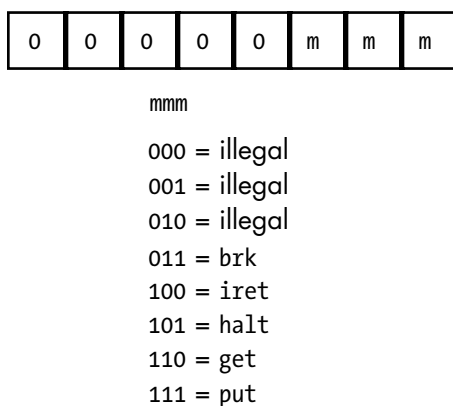


Figure 10-5: Zero-operand instruction encodings (iii = %000 and rr = %00)

The second 2-bit encoding for the rr field, %01, is also part of an expansion opcode that provides all the Y86 jump instructions (see Figure 10-6). The third rr field encoding, %10, is for the not instruction. The fourth rr field encoding is currently unassigned. Any attempt to execute an opcode with an iii field encoding of %000 and an rr field encoding of %11 will halt the processor with an illegal instruction error. As previously discussed, CPU designers often reserve unassigned opcodes like this one so they can extend the instruction set in the future (as Intel did when moving from the 80286 processor to the 80386 or from the 32-bit x86 processors to the 64-bit x86-64 processors).

The seven jump instructions in the Y86 instruction set all take the form `jxx address`; The `jmp` instruction copies the 16-bit address value that follows the opcode into the instruction pointer register, causing the CPU to fetch the next instruction from the target address of the `jmp`. The remaining six instructions—`ja`, `jae`, `jb`, `jbe`, `je`, and `jne`—test some condition and, if it is true, copy the address value into the instruction pointer register. The eighth opcode, %00001111, is another illegal opcode. These encodings are shown in Figure 10-6.

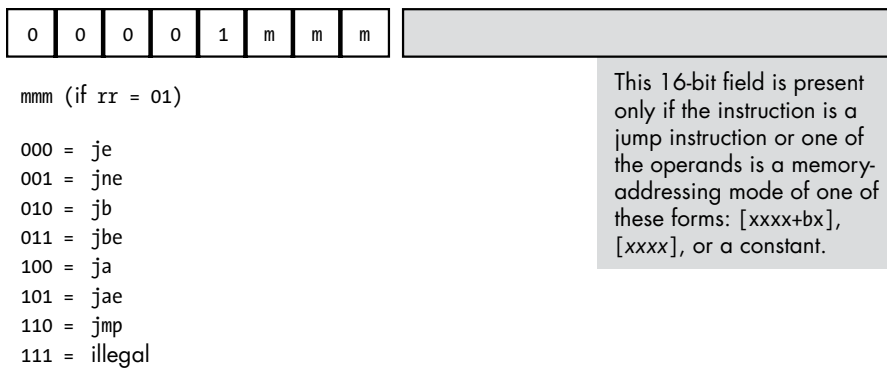


Figure 10-6: Jump instruction encodings

10.3.5 Examples of Encoding Y86 Instructions

The Y86 processor does not execute instructions as human-readable strings of characters like `mov(ax, bx);`. Instead, it fetches instructions as bit patterns, such as `$c1`, from memory, then decodes and executes those bit patterns. Human-readable instructions like `mov(ax, bx);` and `add(5, cx);` must first be converted into binary representation, or *machine code*. This section will explore this conversion.

10.3.5.1 The add Instruction

We'll start our conversion with a very simple example, the `add(cx, dx);` instruction. Once you've chosen the instruction, you look it up in one of the opcode figures from the previous section. The `add` instruction is in the first group (see Figure 10-3) and has an `iii` field of `%101`. The source operand is `cx`, so the `mmm` field is `%010`. The destination operand is `dx`, so the `rr` field is `%11`. Merging these bits produces the opcode `%10111010`, or `$ba` (see Figure 10-7).

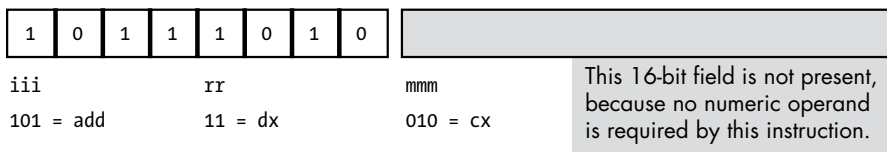


Figure 10-7: Encoding the `add( cx, dx );` instruction

Now consider the `add(5, ax)` instruction. Because it has an immediate source operand (a constant), the `mmm` field will be `%111` (see Figure 10-3). The destination register operand is `ax` (`%00`), and the instruction class field is `%101`, so the full opcode becomes `%10100111`, or `$a7`. However, we're not finished yet. We also have to include the 16-bit constant `$0005` as part of the instruction, with the LO byte of the constant following the opcode, and the HO byte of the constant following its LO byte, because the bytes are arranged in little-endian order. So, the sequence of bytes in memory, from lowest address to highest address, is `$a7`, `$05`, `$00` (see Figure 10-8).

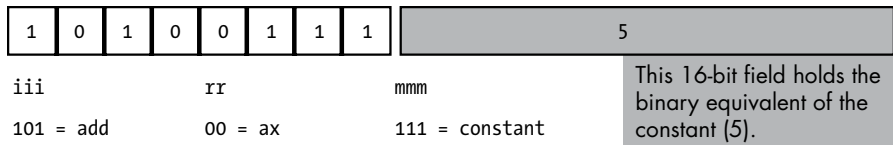


Figure 10-8: Encoding the *add(5, ax);* instruction

The *add([2ff+bx], cx)* instruction also contains a 16-bit constant that is the displacement portion of the indexed addressing mode. To encode this instruction, we use the following field values: *iii* = %101, *rr* = %10, and *mmm* = %101. This produces the opcode byte %10110101, or \$b5. The complete instruction also requires the constant \$2ff, so the full instruction is the 3-byte sequence \$b5, \$ff, \$02 (see Figure 10-9).

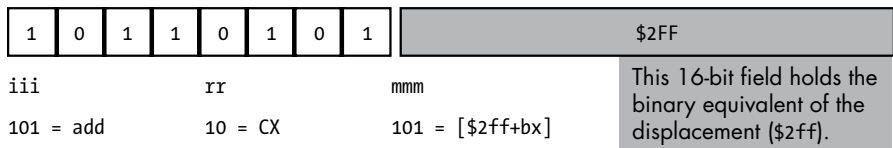


Figure 10-9: Encoding the *add([\$2ff+bx], cx);* instruction

Now consider *add([1000], ax)*. This instruction adds the 16-bit contents of memory locations \$1000 and \$1001 to the value in the AX register. Once again, *iii* = %101 for the *add* instruction. The destination register is *ax*, so *rr* = %00. Finally, the addressing mode is the displacement-only addressing mode, so *mmm* = %110. This forms the opcode %10100110, or \$a6. The complete instruction is 3 bytes long, because it must also encode the displacement (address) of the memory location in the 2 bytes following the opcode. Therefore, the complete 3-byte sequence is \$a6, \$00, \$10 (see Figure 10-10).

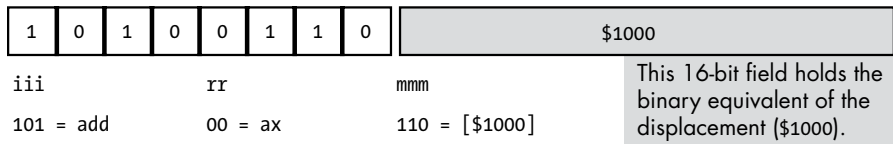


Figure 10-10: Encoding the *add([1000], ax);* instruction

The last addressing mode to consider is the register indirect addressing mode, *[bx]*. The *add([bx], bx)* instruction uses the following encoded values: *mmm* = %101, *rr* = %01 (*bx*), and *mmm* = %100 (*[bx]*). Because the value in the BX register completely specifies the memory address, there's no need to attach a displacement field to the instruction's encoding. Hence, this instruction is only 1 byte long (see Figure 10-11).

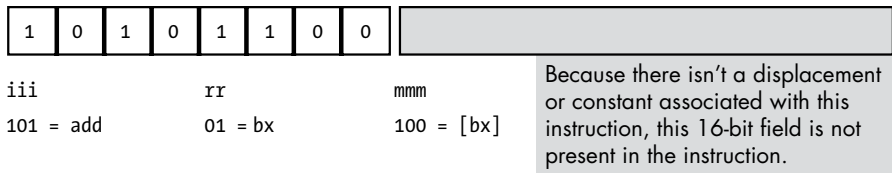


Figure 10-11: Encoding the `add([bx], bx);` instruction

You use a similar approach to encode the `sub`, `cmp`, `and`, and `or` instructions. The only difference between encoding these instructions and the `add` instruction is the value you use for the `iii` field in the opcode.

10.3.5.2 The `mov` Instruction

The Y86 `mov` instruction is special, because it comes in two forms. The only difference between the encoding of the `add` instruction and the encoding of the `mov` instruction's first form (`iii = %110`) is the `iii` field. This form of `mov` copies either a constant or data from the register or memory address specified by the `mmm` field into the destination register specified by the `rr` field.

The second form of the `mov` instruction (`iii = %111`) copies data from the source register specified by the `rr` field to a destination memory location specified by the `mmm` field. In this form of the `mov` instruction, the source and destination meanings of the `rr` and `mmm` fields are reversed: `rr` is the source field and `mmm` is the destination field. Another difference between the two forms of `mov` is that in its second form, the `mmm` field may contain only the values `%100` (`[bx]`), `%101` (`[disp+bx]`), and `%110` (`[disp]`). The destination values can't be any of the registers encoded by `mmm` field values in the range `%000` through `%011` or a constant encoded by an `mmm` field of `%111`. These encodings are illegal because the first form of the `mov` handles cases with a register destination, and because storing data into a constant doesn't make any sense.

10.3.5.3 The `not` Instruction

The `not` instruction is the only instruction with a single memory/register operand that the Y86 processor supports. It has the following syntax:

```
not(reg);
```

or:

```
not(address);
```

where *address* represents one of the memory addressing modes (`[bx]`, `[disp+bx]`, or `[disp]`). You may not specify a constant operand for the `not` instruction.

Because `not` has only a single operand, it needs only the `mmm` field to encode that operand. An `iii` field of `%000` and an `rr` field of `%10` identify the `not` instruction. In fact, whenever the `iii` field contains 0, the CPU knows that it has to decode bits beyond the `iii` field to identify the instruction.

In this case, the *rr* field specifies whether we've encoded *not* or one of the other specially encoded instructions.

To encode an instruction like *not(ax)*, specify %000 for the *iii* field and %10 for the *rr* field, then encode the *mmm* field the same way you would encode it for the *add* instruction. Because *mmm* = %000 for *AX*, *not(ax)* would be encoded as %00010000, or \$10 (see Figure 10-12).

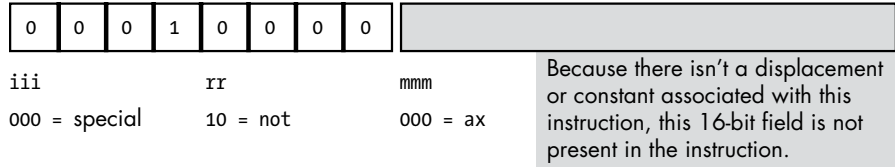


Figure 10-12: Encoding the *not(AX)* instruction

The *not* instruction does not allow an immediate, or constant, operand, so the opcode %00010111 (\$17) is an illegal opcode.

10.3.5.4 The Jump Instructions

The Y86 jump instructions also use the special encoding, meaning that the *iii* field for jump instructions is always %000. These instructions are always 3 bytes long. The first byte, the opcode, specifies which jump instruction to execute, and the next 2 bytes specify the address in memory to which the CPU transfers control (if the condition is met, in the case of the conditional jumps). There are seven different Y86 jump instructions, six conditional jumps, and one unconditional jump, *jmp*. All seven of these instructions set *iii* = %000 and *rr* = %01, so they differ only by their *mmm* fields. The eighth possible opcode, with an *mmm* field value of %111, is an illegal opcode (see Figure 10-6).

Encoding these instructions is relatively straightforward. Picking the instruction you want to encode completely determines the opcode. The opcode values fall in the range \$08 through \$0e (\$0f is the illegal opcode).

The only field that requires some thought is the 16-bit operand that follows the opcode. This field holds the address of the target instruction to which the unconditional jump always transfers, and to which the conditional jumps transfer if the transfer condition is true. To properly encode this 16-bit operand, you must know the address of the opcode byte of the target instruction. If you've already converted the target instruction to binary form and stored it into memory, you're all set—just specify the target instruction's address as the sole operand of the jump instruction. On the other hand, if you haven't yet written, converted, and placed the target instruction into memory, knowing its address would seem to require a bit of divination. Fortunately, you can figure it out by computing the lengths of all the instructions between the current jump instruction you're encoding and the target instruction—but unfortunately, this is an arduous task.

The best way to calculate the distance is to write all your instructions down on paper, compute their lengths (which is easy, because all instructions are either 1 or 3 bytes long depending on whether they have a 16-bit

operand), and then assign an appropriate address to each instruction. Once you've done this, you'll know the starting address for each instruction, and you can put target address operands into your jump instructions as you encode them.

10.3.5.5 The Zero-Operand Instructions

The remaining instructions, the zero-operand instructions, are the easiest to encode. Because they have no operands, they are always 1 byte long. These instructions always have *iii* = %000 and *rr* = %00, and *mmm* specifies the particular instruction opcode (see Figure 10-5). Note that the Y86 CPU leaves five of these instructions undefined (so we can use these opcodes for future expansion).

10.3.6 Extending the Y86 Instruction Set

The Y86 CPU is a trivial CPU, suitable only for demonstrating how to encode machine instructions. However, as with any good CPU, the Y86 design allows for expansion by adding new instructions.

You can extend the number of instructions in a CPU's instruction set by using either undefined or illegal opcodes. So, because the Y86 CPU has several illegal and undefined opcodes, we'll use them to expand the instruction set.

Using undefined opcodes to define new instructions works best when there are undefined bit patterns within an opcode group, and the new instruction you want to add falls into that same group. For example, the opcode %00011*mmm* falls into the same group as the *not* instruction, which also has an *iii* field value of %000. If you decided that you really needed a *neg* (negate) instruction, using the %00011*mmm* opcode makes sense because you'd probably expect *neg* to use the same syntax as the *not* instruction. Likewise, if you want to add a zero-operand instruction to the instruction set, Y86 has five undefined zero-operand instructions for you to choose from (%00000000..%00000100; see Figure 10-5). You'd just appropriate one of these opcodes and assign your instruction to it.

Unfortunately, the Y86 CPU doesn't have many illegal opcodes available. For example, if you wanted to add the *shl* (shift left), *shr* (shift right), *rol* (rotate left), and *ror* (rotate right) instructions as single-operand instructions, there's not enough space within the group of single-operand instruction opcodes to do so (only %00011*mmm* is currently open). Likewise, there are no two-operand opcodes open, so if you wanted to add an *xor* (exclusive OR) instruction or some other two-operand instruction, you'd be out of luck.

A common way to handle this dilemma, and one the Intel designers have employed, is to use one of the undefined opcodes as a prefix opcode byte. For example, the opcode \$ff is illegal (it corresponds to a *mov(dx,*

constant) instruction), but we can use it as a special prefix byte to further expand the instruction set (see Figure 10-13).⁴

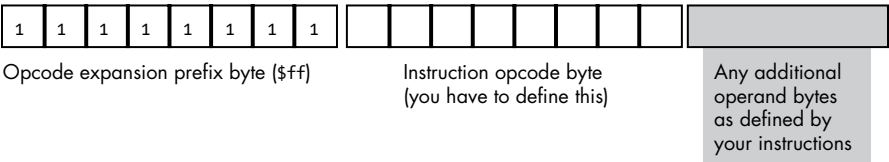


Figure 10-13: Using a prefix byte to extend the instruction set

Whenever the CPU encounters a prefix byte in memory, it reads and decodes the next byte in memory as the actual opcode. However, it doesn't treat the second byte as it would a standard opcode that did not follow a prefix byte. Instead, it allows the CPU designer to create a completely new opcode scheme, independent of the original instruction set. A single-expansion opcode byte allows CPU designers to add up to 256 more instructions to the instruction set. For even more instructions, designers can use additional illegal opcode bytes (in the original instruction set) to add still more expansion opcodes, each with its own independent instruction set; or they can follow the opcode expansion prefix byte with a 2-byte opcode (yielding up to 65,536 new instructions); or they can execute any other scheme they can dream up.

Of course, one big drawback of this approach is that it increases the size of the new instructions by 1 byte, because each instruction now requires the prefix byte as part of the opcode. This also increases the cost of the circuitry (since decoding prefix bytes and multiple instruction sets is fairly complex), so you don't want to use this scheme for the basic instruction set. Nevertheless, it is a good way to expand the instruction set when you've run out of opcodes.

10.4 Encoding 80x86 Instructions

The Y86 processor is simple to understand; we can easily encode instructions by hand for it, and it's a great vehicle for learning how to assign opcodes. It's also a purely hypothetical device intended only as a teaching tool. So, it's time to take a look at the machine instruction format for a real CPU: the 80x86. After all, the programs you write will run on a real CPU, so to fully appreciate what your compilers are doing with your code—so you can choose the best statements and data structures when writing that code—you need to understand how real instructions are encoded.

Even if you're using a different CPU, studying the 80x86 instruction encoding is helpful. They don't call the 80x86 a *complex* instruction set computer (CISC) chip for nothing. Although more complex instruction

4. We could also have used values \$f7, \$ef, and \$e7, as they correspond as well to an attempt to store a register into a constant. However, \$ff is easier to decode. Still, if you need even more prefix bytes for instruction expansion, these three values are available.

encodings do exist, no one would challenge the assertion that it's one of the more complex instruction sets in common use today. Therefore, exploring it will provide valuable insight into the operation of other real-world CPUs.

The generic 80x86 32-bit instruction takes the form shown in Figure 10-14.⁵

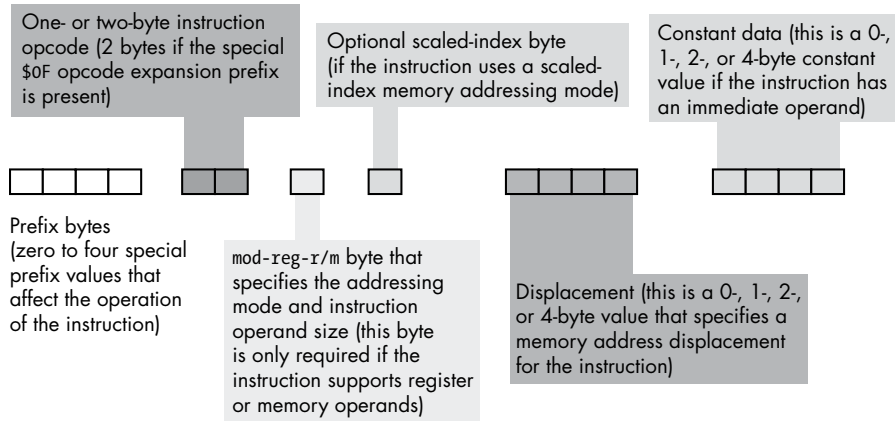


Figure 10-14: 80x86 32-bit instruction encoding

NOTE

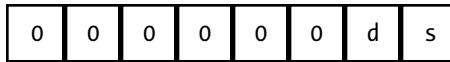
Although this diagram seems to imply that instructions can be up to 16 bytes long, 15 bytes is actually the limit.

The prefix bytes are not the same as the opcode expansion prefix byte that we discussed in the previous section. Instead, the 80x86 prefix bytes modify the behavior of existing instructions. An instruction may have a maximum of four prefix bytes attached to it, but the 80x86 supports more than four different prefix values. The behaviors of many prefix bytes are mutually exclusive, and the results of an instruction will be undefined if you prepend a pair of mutually exclusive prefix bytes to it. We'll take a look at a couple of these prefix bytes in a moment.

The (32-bit) 80x86 supports two basic opcode sizes: a standard 1-byte opcode and a 2-byte opcode consisting of a \$0f opcode expansion prefix byte and a second byte specifying the actual instruction. One way to think of this opcode expansion prefix byte is as an 8-bit extension of the *iii* field in the Y86 encoding. This enables the encoding of up to 512 different instruction classes, although the 80x86 doesn't yet use them all. In reality, various instruction classes use certain bits in this opcode expansion prefix byte for decidedly non-instruction-class purposes. For example, consider the *add* instruction opcode shown in Figure 10-15.

Bit 1 (*d*) specifies the direction of the transfer. If this bit is 0, then the destination operand is a memory location, such as in *add(a1, [ebx]);*. If this bit is 1, the destination operand is a register, as in *add([ebx], a1);*.

5. The 64-bit variants of the 80x86 instruction set complicate things even further.



add opcode

d = 0 if adding from register to memory

d = 1 if adding from memory to register

s = 0 if adding 8-bit operands

s = 1 if adding 16-bit or 32-bit operands

Figure 10-15: 80x86 add opcode

Bit 0 (s) specifies the size of the operands the add instruction operates upon. There's a problem here, however. The 32-bit 80x86 family supports up to three different operand sizes: 8-bit operands, 16-bit operands, and 32-bit operands. With a single size bit, the instruction can encode only two of these three different sizes. In 32-bit operating systems, the vast majority of operands are either 8 bits or 32 bits, so the 80x86 CPU uses the size bit in the opcode to encode those sizes. For 16-bit operands, which occur less frequently than 8-bit or 32-bit operands, Intel uses a special opcode prefix byte to specify the size. As long as instructions that have 16-bit operands occur less than one out of every eight instructions (which is generally the case), this is more compact than adding another bit to the instruction's size. Using a size prefix byte allowed Intel's designers to extend the number of operand sizes without having to change the instruction encoding inherited from the original 16-bit processors in this CPU family.

Note that the AMD/Intel 64-bit architectures go even crazier with opcode prefix bytes. However, the CPU operates in a special 64-bit mode; effectively, the 64-bit 80x86 CPUs (often called the *X86-64 CPUs*) have two completely different instruction sets, each with its own encoding. The X86-64 CPUs can switch between 64- and 32-bit modes to handle programs written in either of the different instruction sets. The encoding in this chapter covers the 32-bit variant; see the Intel or AMD documentation for details on the 64-bit version.

10.4.1 Encoding Instruction Operands

The mod-reg-r/m byte (see Figure 10-14) provides the encoding for instruction operands by specifying the base addressing mode used to access them as well as their size. This byte contains the fields shown in Figure 10-16.

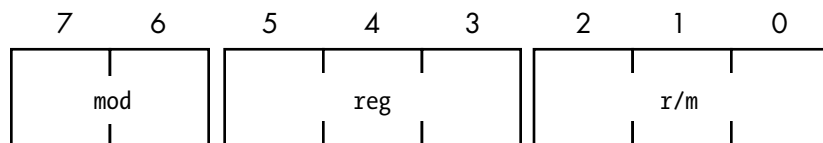


Figure 10-16: mod-reg-r/m byte

The `reg` field almost always specifies an 80x86 register. However, depending on the instruction, the register specified by `reg` can be either the source or the destination operand. To distinguish between the two, many instructions' upcodes include the `d` (direction) field, which contains a value of 0 when `reg` is the source and a value of 1 when it's the destination operand.

This field uses the 3-bit register encodings found in Table 10-1. As just discussed, the size bit in the instruction's opcode indicates whether the `reg` field specifies an 8- or 32-bit register (when operating under a modern, 32-bit operating system). To make the `reg` field specify a 16-bit register, you must set the size bit in the opcode to 1, as well as adding an extra prefix byte.

Table 10-1: `reg` Field Encodings

reg value	Register if data size is 8 bits	Register if data size is 16 bits	Register if data size is 32 bits
%000	al	ax	eax
%001	cl	cx	ecx
%010	dl	dx	edx
%011	bl	bx	ebx
%100	ah	sp	esp
%101	ch	bp	ebp
%110	dh	si	esi
%111	bh	di	edi

With the `d` bit in the opcode of a two-operand instruction indicating whether the `reg` field contains the source or destination operand, the `mod` and `r/m` fields together specify the other of the two operands. In the case of a single-operand instruction like `not` or `neg`, the `reg` field contains an opcode extension, and `mod` and `r/m` combine to specify the only operand. The operand addressing modes specified by the `mod` and `r/m` fields are listed in Tables 10-2 and 10-3.

Table 10-2: `mod` Field Encodings

mod	Description
%00	Specifies register-indirect addressing mode (with two exceptions: scaled-index [<code>sib</code>] addressing modes with no displacement operand when <code>r/m</code> = %100; and displacement-only addressing mode when <code>r/m</code> = %101).
%01	Specifies that a 1-byte signed displacement follows the addressing mode byte(s).
%10	Specifies that a 1-byte signed displacement follows the addressing mode byte(s).
%11	Specifies direct register access.

Table 10-3: mod-r/m Encodings

mod	r/m	Addressing mode
%00	%000	[eax]
%01	%000	[eax+disp ₈]
%10	%000	[eax+disp ₃₂]
%11	%000	al, ax, or eax
%00	%001	[ecx]
%01	%001	[ecx+disp ₈]
%10	%001	[ecx+disp ₃₂]
%11	%001	cl, cx, or ecx
%00	%010	[edx]
%01	%010	[edx+disp ₈]
%10	%010	[edx+disp ₃₂]
%11	%010	dl, dx, or edx
%00	%011	[ebx]
%01	%011	[ebx+disp ₈]
%10	%011	[ebx+disp ₃₂]
%11	%011	bl, bx, or ebx
%00	%100	Scaled-index (sib) mode
%01	%100	sib + disp ₈ mode
%10	%100	sib + disp ₃₂ mode
%11	%100	ah, sp, or esp
%00	%101	Displacement-only mode (32-bit displacement)
%01	%101	[ebp+disp ₈]
%10	%101	[ebp+disp ₃₂]
%11	%101	ch, bp, or ebp
%00	%110	[esi]
%01	%110	[esi+disp ₈]
%10	%110	[esi+disp ₃₂]
%11	%110	dh, si, or esi
%00	%111	[edi]
%01	%111	[edi+disp ₈]
%10	%111	[edi+disp ₃₂]
%11	%111	bh, di, or edi

There are a couple of interesting things to note about Tables 10-2 and 10-3. First, there are two different forms of the $[reg+disp]$ addressing modes: one form with an 8-bit displacement and one form with a 32-bit displacement. Addressing modes whose displacement falls in the range -128 through $+127$ require only a single byte after the opcode to encode the displacement. Instructions with a displacement that falls within this range will be shorter and sometimes faster than instructions whose displacement values are not within this range and thus require 4 bytes after the opcode.

The second thing to note is that there is no $[ebp]$ addressing mode. If you look at the entry in Table 10-3 where this addressing mode logically belongs (where r/m is $\%101$ and mod is $\%00$), you'll find that its slot is occupied by the 32-bit displacement-only addressing mode. The basic encoding scheme for addressing modes didn't allow for a displacement-only addressing mode, so Intel "stole" the encoding for $[ebp]$ and used that for the displacement-only mode. Fortunately, anything you can do with the $[ebp]$ addressing mode you can also do with the $[ebp+disp_8]$ addressing mode by setting the 8-bit displacement to 0. While such an instruction is a bit longer than it would otherwise need to be if the $[ebp]$ addressing mode existed, the same capabilities are still there. Intel wisely chose to replace this particular register-indirect addressing mode, anticipating that programmers would use it less often than the other register-indirect addressing modes.

Another thing you'll notice missing from this table are addressing modes of the form $[esp]$, $[esp+disp_8]$, and $[esp+disp_{32}]$. Intel's designers borrowed the encodings for these three addressing modes to support the *scaled-index addressing* modes they added to their 32-bit processors in the 80x86 family.

If $r/m = \%100$ and $mod = \%00$, this specifies an addressing mode of the form $[reg_1 32 + reg_2 32 * n]$. This scaled-index addressing mode computes the final address in memory as the sum of reg_2 multiplied by n ($n = 1, 2, 4$, or 8) and reg_1 . Programs most often use this addressing mode when reg_1 is a pointer holding the base address of an array of bytes ($n = 1$), words ($n = 2$), double words ($n = 4$), or quad words ($n = 8$), and reg_2 holds the index into that array.

If $r/m = \%100$ and $mod = \%01$, this specifies an addressing mode of the form $[reg_1 32 + reg_2 32 * n + disp_8]$. This scaled-index addressing mode computes the final address in memory as the sum of reg_2 multiplied by n ($n = 1, 2, 4$, or 8), reg_1 , and the 8-bit signed displacement (sign-extended to 32 bits). Programs most often use this addressing mode when reg_1 is a pointer holding the base address of an array of records, reg_2 holds the index into that array, and $disp_8$ provides the offset to a desired field in the record.

If $r/m = \%100$ and $mod = \%10$, this specifies an addressing mode of the form $[reg_1 32 + reg_2 32 * n + disp_{32}]$. This scaled-index addressing mode computes the

final address in memory as the sum of reg_2 multiplied by n ($n = 1, 2, 4, \text{ or } 8$), reg_1 , and the 32-bit signed displacement. Programs most often use this addressing mode to index into static arrays of bytes, words, double words, or quad words.

If values corresponding to one of the `sib` modes appear in the `mod` and `r/m` fields, the addressing mode is a scaled-index addressing mode with a second byte (the `sib`) following the `mod-reg-r/m` byte, though don't forget that the `mod` field still specifies a displacement size of 0, 1, or 4 bytes. Figure 10-17 shows the layout of this extra `sib`, and Tables 10-4, 10-5, and 10-6 explain the values for each of the `sib` fields.

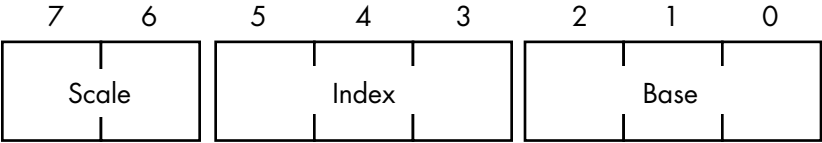


Figure 10-17: The `sib` (scaled-index byte) layout

Table 10-4: Scale Values

Scale value	Index * scale value
%00	Index * 1
%01	Index * 2
%10	Index * 4
%11	Index * 8

Table 10-5: Register Values for `sib` Encoding

Index value	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
%100	Illegal
%101	EBP
%110	ESI
%111	EDI

Table 10-6: Base Register Values for sib Encoding

Base value	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
%100	ESP
%101	Displacement only if mod = %00, EBP if mod = %01 or %10
%110	ESI
%111	EDI

The mod-reg-r/m and sib bytes are complex and convoluted, no question about that. The reason is that Intel reused its 16-bit addressing circuitry when it switched to the 32-bit format rather than simply abandoning it at that point. There were good hardware reasons for retaining it, but the result is a complex scheme for specifying addressing modes. As you can imagine, things got even worse when Intel and AMD developed the x86-64 architecture.

Note that if the r/m field of the mod-reg-r/m byte contains %100 and mod does not contain %11, the addressing mode is a sib mode rather than the expected $[esp]$, $[esp+disp_8]$, or $[esp+disp_{32}]$ mode. In this case the compiler or assembler will emit an extra sib byte immediately following the mod-reg-r/m byte. Table 10-7 lists the various combinations of legal scaled-index addressing modes on the 80x86.

In each of the addressing modes listed in Table 10-7, the mod field of the mod-reg-r/m byte specifies the size of the displacement (0, 1, or 4 bytes). The base and index fields of the sib specify the base and index registers, respectively. Note that this addressing mode does not allow the use of ESP as an index register. Presumably, Intel left this particular mode undefined to allow for extending the addressing modes to 3 bytes in a future version of the CPU, although doing so seems a bit extreme.

Just as the mod-reg-r/m encoding replaced the $[ebp]$ addressing mode with a displacement-only mode, the sib addressing format replaces the $[ebp+index*scale]$ mode with a displacement-plus index mode (that is, no base register). If it turns out that you really need to use the $[ebp+index*scale]$ addressing mode, you'll have to use the $[disp_8+ebp+index*scale]$ mode instead, specifying a 1-byte displacement value of 0.

Table 10-7: The Scaled-Index Addressing Modes

mod	Index	Legal scaled-index addressing modes ⁶
%00 Base ≠ %101	%000	$[base_{32}+eax*n]$
	%001	$[base_{32}+ecx*n]$
	%010	$[base_{32}+edx*n]$
	%011	$[base_{32}+ebx*n]$
	%100	n/a ⁷
	%101	$[base_{32}+ebp*n]$
	%110	$[base_{32}+esi*n]$
	%111	$[base_{32}+edi*n]$
%00 Base = %101 ⁸	%000	$[disp_{32}+eax*n]$
	%001	$[disp_{32}+ecx*n]$
	%010	$[disp_{32}+edx*n]$
	%011	$[disp_{32}+ebx*n]$
	%100	n/a
	%101	$[disp_{32}+ebp*n]$
	%110	$[disp_{32}+esi*n]$
	%111	$[disp_{32}+edi*n]$
%01	%000	$[disp_8+base_{32}+eax*n]$
	%001	$[disp_8+base_{32}+ecx*n]$
	%010	$[disp_8+base_{32}+edx*n]$
	%011	$[disp_8+base_{32}+ebx*n]$
	%100	n/a
	%101	$[disp_8+base_{32}+ebp*n]$
	%110	$[disp_8+base_{32}+esi*n]$
	%111	$[disp_8+base_{32}+edi*n]$
%10	%000	$[disp_{32}+base_{32}+eax*n]$
	%001	$[disp_{32}+base_{32}+ecx*n]$
	%010	$[disp_{32}+base_{32}+edx*n]$
	%011	$[disp_{32}+base_{32}+ebx*n]$
	%100	n/a
	%101	$[disp_{32}+base_{32}+ebp*n]$
	%110	$[disp_{32}+base_{32}+esi*n]$
	%111	$[disp_{32}+base_{32}+edi*n]$

6. The $base_{32}$ register can be any of the 80x86 32-bit general-purpose registers, as specified by the base field.

7. The 80x86 does not allow a program to use the ESP as an index register.

8. The 80x86 doesn't support a $[base_{32}+ebp*n]$ addressing mode, but you can achieve the same effective address using $[base_{32}+ebp*n+disp_8]$ with an 8-bit displacement value of 0.

10.4.2 Encoding the add Instruction

To help you figure out how to encode an instruction using this complex scheme, let's look at an example of the 80x86 add instruction using various addressing modes. The add opcode is either \$00, \$01, \$02, or \$03, depending on its direction and size bits (see Figure 10-15). Figures 10-18 through 10-25 show how to encode various forms of the add instruction using different addressing modes.

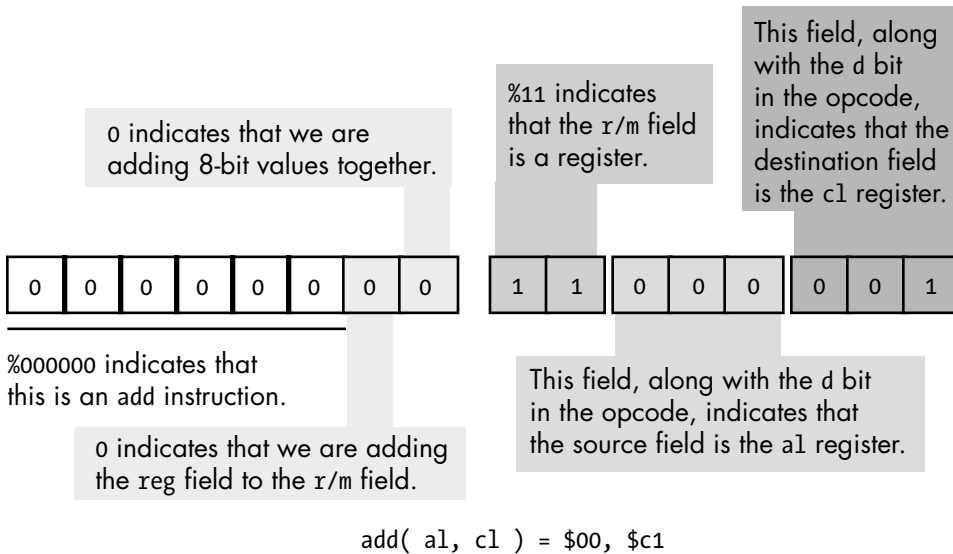


Figure 10-18: Encoding the `add(a1, c1)` instruction

There is an interesting side effect of the mod-reg-r/m organization and direction bit: some instructions have two different legal opcodes. For example, we could also encode the `add(a1, c1)` instruction shown in Figure 10-18 as `$02, $c8` by reversing the positions of the AL and CL registers in the reg and r/m fields and then setting the d bit (bit 1) in the opcode to 1. This applies to all instructions with two register operands and a direction bit, such as the `add(eax, ecx)` instruction in Figure 10-19, which can also be encoded as `$03, $c8`.

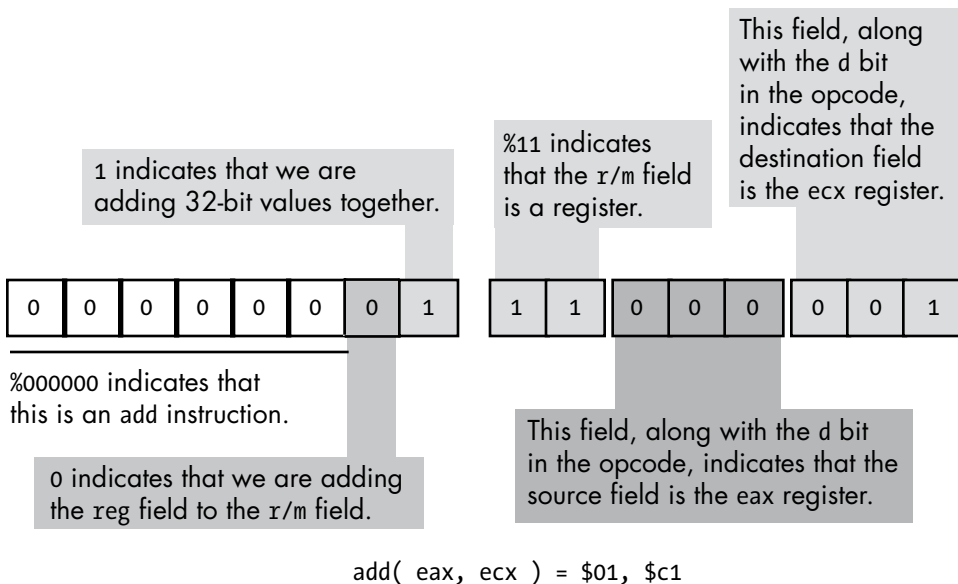


Figure 10-19: Encoding the `add(eax, ecx);` instruction

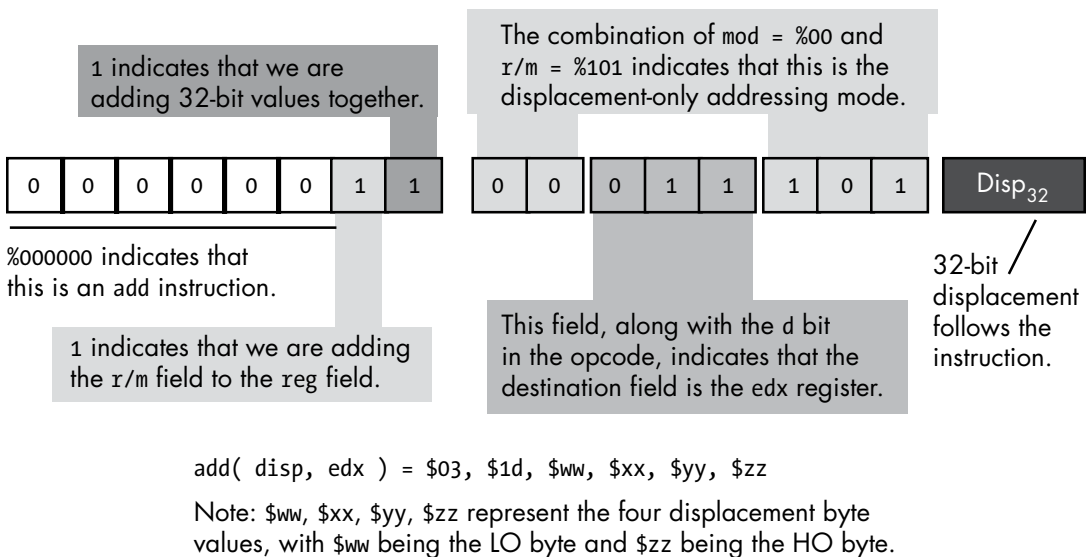
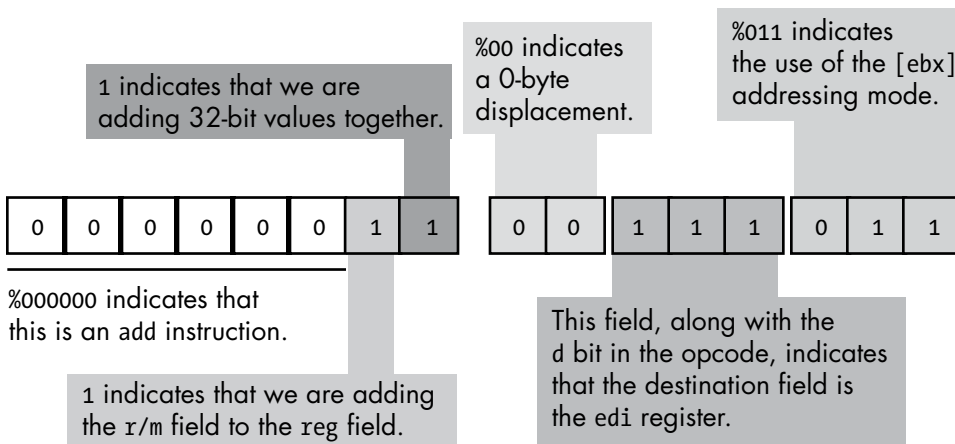
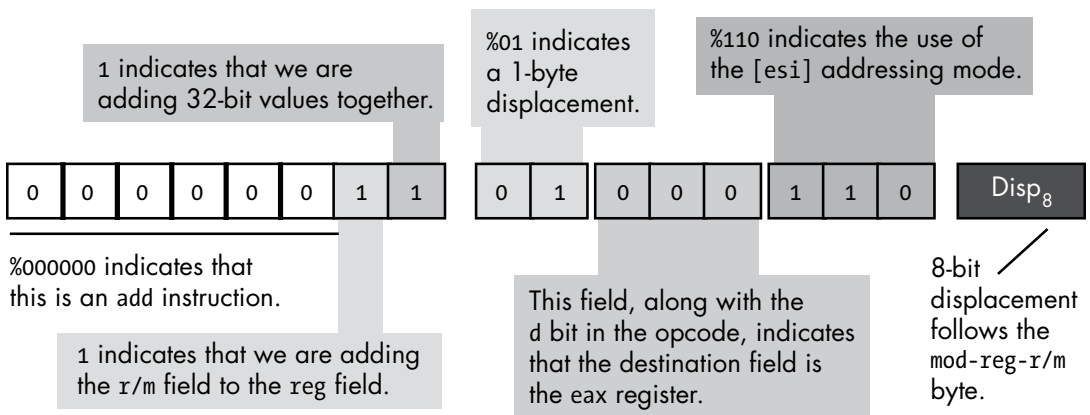


Figure 10-20: Encoding the `add(displacement, edx);` instruction



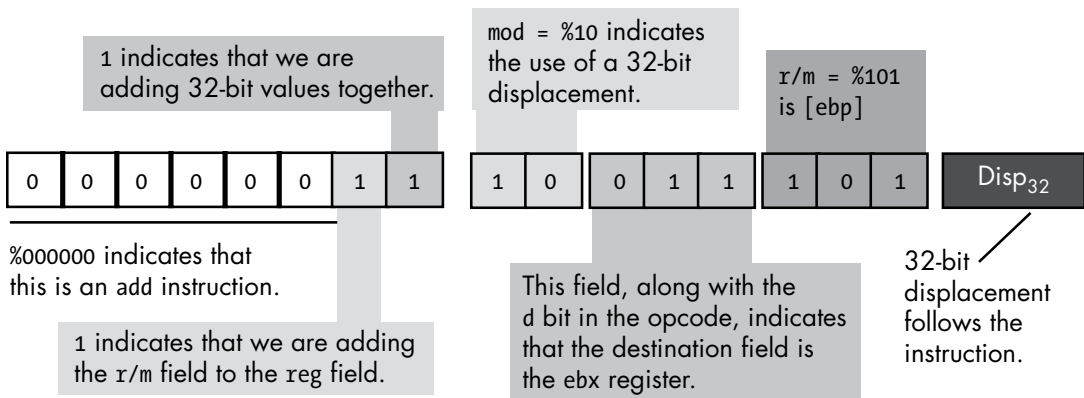
`add( [ebx], edi ) = $03, $3b`

Figure 10-21: Encoding the `add([ebx], edi);` instruction



`add( [esi + disp8 ], eax ) = $03, $46, $xx`

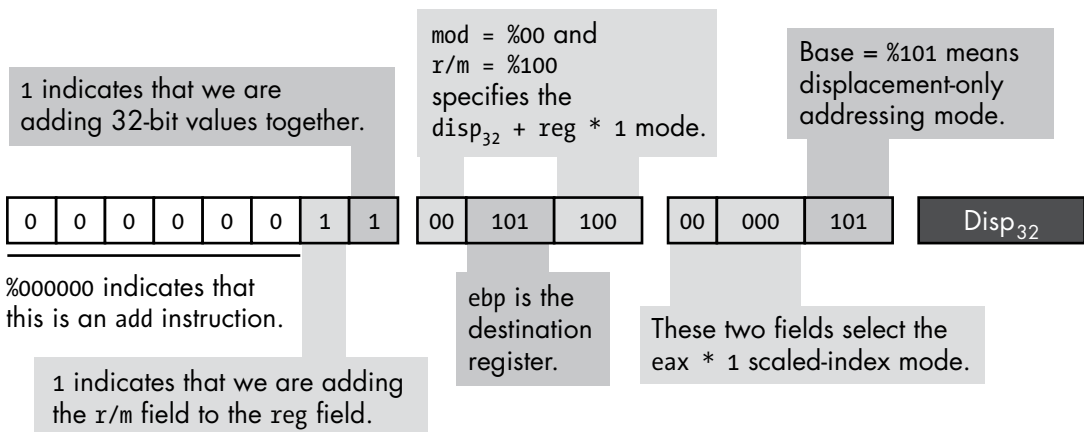
Figure 10-22: Encoding the `add([esi+disp8], eax);` instruction



$\text{add}([ebp + \text{disp}_{32}], ebx) = \$03, \$9d, \$ww, \$xx, \$yy, \$zz$

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values, with \$ww being the LO byte and \$zz being the HO byte.

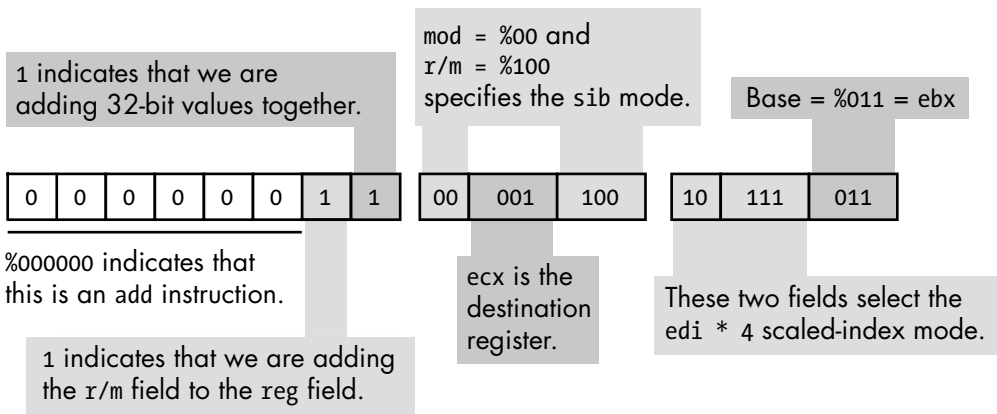
Figure 10-23: Encoding the $\text{add}([ebp + \text{disp}_{32}], ebx)$ instruction



$\text{add}([\text{disp}_{32} + \text{eax} * 1], ebp) = \$03, \$2c, \$05, \$ww, \$xx, \$yy, \zz

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values, with \$ww being the LO byte and \$zz being the HO byte.

Figure 10-24: Encoding the $\text{add}([\text{disp}_{32} + \text{eax} * 1], ebp)$ instruction



`add( [ebx + edi * 4], ecx ) = $03, $0c, $bb`

Figure 10-25: Encoding the `add([ebx+edi*4], ecx)`; instruction

10.4.3 Encoding Immediate (Constant) Operands on the x86

You may have noticed that the mod-reg-r/m and sib bytes don't contain any bit combinations you can use to specify that an instruction contains an immediate operand. The 80x86 uses a completely different opcode to specify an immediate operand. Figure 10-26 shows the basic encoding for an add immediate instruction.

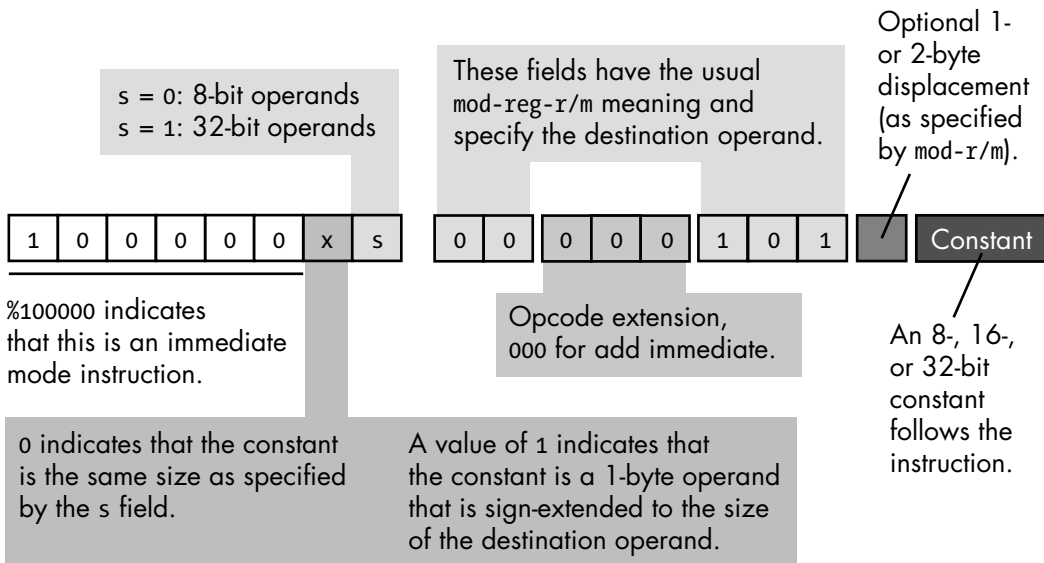


Figure 10-26: Encoding an add immediate instruction

There are three major differences between the encoding of the add immediate instruction and the standard add instruction. First, and most important, the opcode has a 1 in the HO bit position. This tells the CPU that the instruction has an immediate constant. This change alone, however, does not tell the CPU that it must execute an add instruction, as you'll see shortly.

The second difference is that there's no direction bit in the opcode. This makes sense because you cannot specify a constant as a destination operand. Therefore, the destination operand is always the location specified by the mod and r/m bits in the mod-reg-r/m field.

In place of the direction bit, the opcode has a sign-extension (x) bit. For 8-bit operands, the CPU ignores the sign-extension bit. For 16-bit and 32-bit operands, the sign-extension bit specifies the size of the constant following the add instruction. If the sign-extension bit contains 0, the constant is already the same size as the operand (either 16 or 32 bits). If the sign-extension bit contains 1, the constant is a signed 8-bit value, and the CPU sign-extends this value to the appropriate size before adding it to the operand. This little trick often makes programs much shorter, because you commonly add small constants to 16- or 32-bit destination operands.

The third difference between the add immediate and the standard add instruction is the meaning of the reg field in the mod-reg-r/m byte. Because the instruction implies that the source operand is a constant, and the mod-r/m fields specify the destination operand, the instruction does not need to use the reg field to specify an operand. Instead, the 80x86 CPU uses these 3 bits as an opcode extension. For the add immediate instruction, these 3 bits must contain 0, and another bit pattern would correspond to a different instruction.

When a constant is added to a memory location, any displacement associated with that memory location immediately precedes the constant data in the instruction sequence.

10.4.4 Encoding 8-, 16-, and 32-Bit Operands

When designing the 8086, Intel used one opcode bit (s) to specify whether the operand sizes were 8 or 16 bits. Later, when it extended the 80x86 architecture to 32 bits with the introduction of the 80386, Intel had a problem: with this single operand size bit, it could encode only two sizes, but it needed to encode three (8, 16, and 32 bits). To solve this problem, Intel used an *operand-size prefix byte*.

Intel studied its instruction set and concluded that in a 32-bit environment, programs were likely to use 8-bit and 32-bit operands far more often than 16-bit operands. Therefore, it decided to let the size bit (s) in the opcode select between 8- and 32-bit operands, as described in the previous sections. Although modern 32-bit programs don't use 16-bit operands very often, they do need them now and then. So, Intel lets you prefix a 32-bit instruction with the operand-size prefix byte, whose value is \$66, and this prefix byte tells the CPU that the operands contain 16-bit data rather than 32-bit data.

You do not have to explicitly add an operand-size prefix byte to your 16-bit instructions; the assembler or compiler takes care of this automatically for you. However, do keep in mind that whenever you use a 16-bit object in a 32-bit program, the instruction is 1 byte longer because of the prefix value. Therefore, you should be careful about using 16-bit instructions if size and, to a lesser extent, speed are important.

10.4.5 Encoding 64-Bit Operands

When running in 64-bit mode, Intel and AMD x84-64 processors use special opcode prefix bytes to specify 64-bit registers. There are 16 REX opcode bytes that handle 64-bit operands and addressing modes. Because there weren't 16 single-byte opcodes available, AMD (who designed the instruction set) chose to repurpose 16 existing opcodes (the 1-byte opcode variants for the `inc(reg)` and `dec(reg)` instructions). There are still 2-byte variants of these instructions, so rather than eliminating the instructions altogether, AMD just removed the 1-byte versions. However, standard 32-bit code (a lot of which certainly uses those 1-byte increment and decrement instructions) can no longer run on the 64-bit model. That's why AMD and Intel introduced new 32-bit and 64-bit operation modes—so the CPUs could run both older 32-bit code and newer 64-bit code on the same piece of silicon.

10.4.6 Alternate Encodings for Instructions

As noted earlier in this chapter, one of Intel's primary design goals for the 80x86 was to create an instruction set that allowed programmers to write very short programs in order to save memory, which was precious at the time. One way Intel did this was to create alternative encodings of some very commonly used instructions. These alternative instructions were shorter than their standard counterparts, and Intel hoped that programmers would make extensive use of the shorter versions, thereby creating shorter programs.

A good example of these alternative instructions are the `add(constant, accumulator)`; instructions, where the accumulator is `al`, `ax`, or `eax`. The 80x86 provides 1-byte opcodes for `add(constant, al)`; and `add(constant, eax)`; which are \$04 and \$05, respectively. With a 1-byte opcode and no `mod-reg-r/m` byte, these instructions are 1 byte shorter than their standard `add immediate` counterparts. The `add(constant, ax)`; instruction requires an operand-size prefix, so its opcode is effectively 2 bytes. However, this is still 1 byte shorter than the corresponding standard `add immediate`.

You don't have to specify anything special to use these instructions. Any decent assembler or compiler will automatically choose the shortest possible instruction it can use when translating your source code into machine code. However, you should note that Intel provides alternative encodings only for the accumulator registers. Therefore, if you have a choice of several instructions to use and the accumulator registers are among these choices, the `AL`, `AX`, and `EAX` registers are often your best bet. However, this option is usually available only to assembly language programmers.

10.5 Implications of Instruction Set Design to the Programmer

Only by knowing the computer’s architecture and, in particular, how the CPU encodes machine instructions, can you make the most efficient use of the machine’s instructions. By studying instruction set design, you can gain a clear understanding of the following:

- Why some instructions are shorter than others
- Why some instructions are faster than others
- Which constant values the CPU can handle efficiently
- Whether constants are more efficient than memory locations
- Why certain arithmetic and logical operations are more efficient than others
- Which types of arithmetic expressions are more easily translated into machine code than other types
- Why code is less efficient if it transfers control over a large distance in the object code

. . . and so on.

By studying instruction set design, you become more aware of the implications of the code you write (even in an HLL) in terms of efficient operation on the CPU. Armed with this knowledge, you’ll be better equipped to write great code.

10.6 For More Information

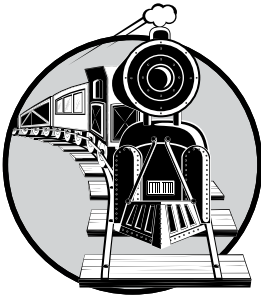
Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Elsevier, 2012.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

Intel. “Intel® 64 and IA-32 Architectures Software Developer Manuals.” Last updated November 11, 2019. <https://software.intel.com/en-us/articles/intel-sdm/>.

11

MEMORY ARCHITECTURE AND ORGANIZATION



This chapter discusses memory hierarchy—the different types and performance levels of memory found in computer systems.

Although programmers often treat all forms of memory as though they are equivalent, using memory improperly can have a negative impact on performance. In this chapter you'll see how to make the best use of the memory hierarchy within your programs.

11.1 The Memory Hierarchy

Most modern programs benefit by having a large amount of very fast memory. Unfortunately, as a memory device gets larger, it tends to be slower. For example, cache memories are very fast, but they are also small and expensive. Main memory is inexpensive and large, but it is slow, requiring

wait states. The memory hierarchy provides a way to compare the cost and performance of different types of memory. Figure 11-1 shows one variant of the memory hierarchy.

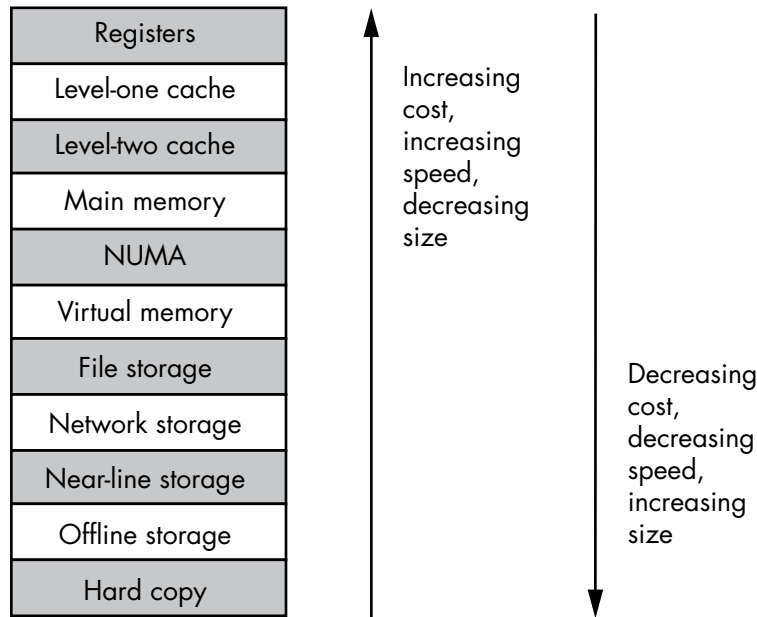


Figure 11-1: The memory hierarchy

At the top level of the memory hierarchy are the CPU's general-purpose *registers*. Registers provide the fastest access to data possible on the CPU. The register file is also the smallest memory object in the hierarchy (for example, the 32-bit 80x86 has just eight general-purpose registers, and the x86-64 variants have up to 16 general-purpose registers). Because it is impossible to add more registers to a CPU, they are also the most expensive memory locations. Even if we count the FPU, MMX/AltiVec/Neon, SSE/SIMD, AVX/2/-512, and other CPU registers in this portion of the memory hierarchy, it does not change the fact that CPUs have a very limited number of registers, and the cost per byte of register memory is quite high.

Working our way down, the *level-one (L1) cache* system is the next highest performance subsystem in the memory hierarchy. As with registers, the CPU manufacturer usually provides the L1 cache on the chip, and you cannot expand it. Its size is usually small, typically between 4KB and 32KB, though this is much larger than the register memory available on the CPU chip. Although the L1 cache size is fixed on the CPU, the cost per cache byte is much lower than the cost per register byte, because the cache contains more storage than is available in all the registers combined, and the system designer's cost for both memory types equals the price of the CPU.

Level-two (L2) cache is present on some CPUs, but not all. For example, Intel i3, i5, i7, and i9 CPUs include an L2 cache as part of their package, but some of Intel's older Celeron chips do not. The L2 cache is generally much

larger than the L1 cache (for example, 256KB to 1MB as compared with 4KB to 32KB). On CPUs with a built-in L2 cache, the cache is not expandable. It still costs less than the L1 cache, because we amortize the cost of the CPU across all the bytes in the two caches, and the L2 cache is larger.

Level-three (L3) cache is present on all but the oldest Intel processors. The L3 cache is larger still than the L2 cache (typically 8MB on later Intel chips).

The *main-memory* subsystem comes below the L3 (or L2, if there is no L3) cache system in the memory hierarchy. Main memory is the general-purpose, relatively low-cost memory—typically DRAM or something similarly inexpensive—found in most computer systems. However, there are many differences in main-memory technology that result in variations in speed. The main-memory types include standard DRAM, synchronous DRAM (SDRAM), double data rate DRAM (DDR), DDR3, DDR4, and so on. Generally, you won't find a mixture of these technologies in the same computer system.

Below main memory is the *NUMA* memory subsystem. NUMA, which stands for *Non-Uniform Memory Access*, is a bit of a misnomer. The term implies that different types of memory have different access times, which describes the entire memory hierarchy; in Figure 11-1, however, it refers to blocks of memory that are electronically similar to main memory but, for one reason or another, operate significantly slower. A good example of NUMA is the memory on a video (or graphics) card. Another example is flash memory, which has significantly slower access and transfer times than standard semiconductor RAM. Other peripheral devices that provide a block of memory to be shared between the CPU and the peripheral usually have slow access times as well.

Most modern computer systems implement a *virtual memory* scheme that simulates main memory using a mass storage disk drive. A virtual memory subsystem is responsible for transparently copying data between the disk and main memory as programs need it. While disks are significantly slower than main memory, the cost per bit is also three orders of magnitude lower for disks. Therefore, it's far less expensive to keep data on magnetic storage or on a solid-state drive (SSD) than in main memory.

File storage memory also uses disk media to store program data. However, whereas the virtual memory subsystem is responsible for transferring data between disk (or SSD) and main memory as programs require, it is the program's responsibility to store and retrieve file storage data. In many instances, it's a bit slower to use file storage memory than it is to use virtual memory, which is why file storage memory is lower in the memory hierarchy.¹

Next comes *network storage*. At this level in the memory hierarchy, programs keep data on a different memory system that connects to the computer system via a network. Network storage can be virtual memory, file storage memory, or *distributed shared memory (DSM)*, where processes running on different computer systems share data stored in a common block of memory and communicate changes to that block across the network.

1. Note, however, that in some degenerate cases virtual memory can be much slower than file access.

Virtual memory, file storage, and network storage are examples of *online memory subsystems*. Memory access within these memory subsystems is slower than accessing main memory. However, when a program requests data from one of these three online memory subsystems, the memory device will respond to the request as quickly as its hardware allows. This is not true for the remaining levels in the memory hierarchy.

The *near-line* and *offline storage* subsystems may not be ready to respond immediately to a program's request for data. An offline storage system keeps its data in electronic form (usually magnetic or optical) but on storage media that are not necessarily connected to the computer system that needs the data. Examples of offline storage include magnetic tapes, unattached external disk drives, disk cartridges, optical disks, USB memory sticks, SD cards, and floppy diskettes. When a program needs to access data stored offline, it must stop and wait for someone or something to mount the appropriate media on the computer system. This delay can be quite long (perhaps the computer operator decided to take a coffee break?).

Near-line storage uses the same types of media as offline storage, but rather than requiring an external source to mount the media before its data is available for access, the near-line storage system holds the media in a special robotic jukebox device that can automatically mount the desired media when a program requests it.

Hardcopy storage is simply a printout, in one form or another, of data. If a program requests some data, and that data exists only in hardcopy form, someone will have to manually enter the data into the computer. Paper or other hardcopy media is probably the least expensive form of memory, at least for certain data types.

11.2 How the Memory Hierarchy Operates

The whole point of the memory hierarchy is to allow reasonably fast access to a large amount of memory. If only a little memory were necessary, we'd use fast static RAM (the circuitry that cache memory uses) for everything. If speed wasn't an issue, we'd use virtual memory for everything. The memory hierarchy enables us to take advantage of the principles of *spatial locality of reference* and *temporality of reference* to move frequently referenced data into fast memory and leave rarely referenced data in slower memory. Unfortunately, during the course of a program's execution, the sets of oft-used and seldom-used data change. We can't simply distribute our data throughout the various levels of the memory hierarchy when the program starts and then leave the data alone as the program executes. Instead, the different memory subsystems need to be able to accommodate changes in spatial locality or temporality of reference during the program's execution by dynamically moving data between subsystems.

Moving data between the registers and memory is strictly a program function. The program loads data into registers and stores register data into memory using machine instructions like `mov`. It is the programmer's

or compiler's responsibility to keep heavily referenced data in the registers as long as possible; the CPU will not automatically place data in general-purpose registers in order to achieve higher performance.

Programs explicitly control access to registers, main memory, and those memory-hierarchy subsystems only at the file storage level and below. Programs are largely unaware of the memory hierarchy between the register level and main memory. In particular, cache access and virtual memory operations are generally transparent to the program; that is, access to these levels of the memory hierarchy usually occurs without any intervention on a program's part. Programs simply access main memory, and the hardware and operating system take care of the rest.

Of course, if a program always accesses main memory, it will run slowly, because modern DRAM main-memory subsystems are much slower than the CPU. The job of the cache memory subsystems and of the CPU's cache controller is to move data between main memory and the L1, L2, and L3 caches so that the CPU can quickly access oft-requested data. Likewise, it is the virtual memory subsystem's responsibility to move oft-requested data from hard disk to main memory (if even faster access is needed, the caching subsystem will then move the data from main memory to cache).

With few exceptions, most memory subsystem accesses take place transparently between one level of the memory hierarchy and the level immediately below or above it. For example, the CPU rarely accesses main memory directly. Instead, when the CPU requests data from memory, the L1 cache subsystem takes over. If the requested data is in the cache, the L1 cache subsystem returns the data to the CPU, and that concludes the memory access. If the requested data isn't present in the L1 cache, the L1 cache subsystem passes the request down to the L2 cache subsystem. If the L2 cache subsystem has the data, it returns this data to the L1 cache, which then returns the data to the CPU. Requests for the same data in the near future will be fulfilled by the L1 cache rather than the L2 cache, because the L1 cache now has a copy of the data. After the L2 cache, the L3 cache kicks in.

If none of the L1, L2, or L3 cache subsystems have a copy of the data, the request goes to main memory. If the data is found in main memory, the main-memory subsystem passes it to the L3 cache, which then passes it to the L2 cache, which then passes it to the L1 cache, which then passes it to the CPU. Once again, the data is now in the L1 cache, so any requests for this data in the near future will be fulfilled by the L1 cache.

If the data is not present in main memory but exists in virtual memory on some storage device, the operating system takes over, reads the data from disk or some other device (such as a network storage server), and passes the data to the main-memory subsystem. Main memory then passes the data through the caches to the CPU as previously described.

Because of spatial locality and temporality, the largest percentage of memory accesses takes place in the L1 cache subsystem. The next largest percentage of accesses takes place in the L2 cache subsystem. After that, the L3 cache system handles most accesses. The most infrequent accesses take place in virtual memory.

11.3 Relative Performance of Memory Subsystems

Looking again at Figure 11-1, notice that the speed of the various memory hierarchy levels increases as you go up. Exactly how much faster is each successive level in the memory hierarchy? The short answer is that the speed gradient isn't uniform. The speed difference between any two contiguous levels ranges from "almost no difference" to "four orders of magnitude."

Registers are, unquestionably, the best place to store data you need to access quickly. Accessing a register never requires any extra time, and most machine instructions that access data can access register data. Furthermore, instructions that access memory often require extra bytes (displacement bytes) as part of the instruction encoding. This makes instructions longer and, often, slower.

Intel's instruction timing tables for the 80x86 claim that an instruction like `mov(someVar, ecx)`; should run as fast as an instruction like `mov(ebx, ecx)`;. However, if you read the fine print, you'll find that Intel makes this claim based on several assumptions about the former instruction. First, it assumes that *someVar*'s value is present in the L1 cache memory. If it is not, the cache controller has to look in the L2 cache, in the L3 cache, in main memory, or, worse, on disk in the virtual memory subsystem. All of a sudden, an instruction that should execute in 0.25 nanoseconds on a 4 GHz processor (that is, in one clock cycle) requires several milliseconds to execute. That's a difference of over six orders of magnitude. It's true that future accesses of this variable will take place in just one clock cycle because it will subsequently be stored in the L1 cache. But even if you access *someVar*'s value one million times while it's still in the cache, the average time of each access will still be about two cycles because of how long it takes to access *someVar* the very first time.

Granted, the likelihood that some variable will be located on disk in the virtual memory subsystem is quite low. However, there's still a difference in performance of a couple orders of magnitude between the L1 cache subsystem and the main-memory subsystem. Therefore, if the program has to retrieve the data from main memory, 999 memory accesses later, you're still paying an average cost of two clock cycles to access data that Intel's documentation claims should take one cycle.

The difference in speed between the L1, L2, and L3 cache systems isn't so dramatic unless the secondary or tertiary cache is not packaged together on the CPU. On a 4 GHz processor, the L1 cache must respond within 0.25 nanoseconds if the cache operates with zero wait states (some processors actually introduce wait states in L1 cache accesses, but CPU designers try to avoid this). Accessing data in the L2 cache is always slower than in the L1 cache, and always includes the equivalent of at least one wait state, and probably more.

There are several reasons why L2 cache accesses are slower than L1 accesses. First, it takes the CPU time to determine that the data it's seeking is not in the L1 cache. By the time it does that, the memory access cycle is nearly complete, and there's no time to access the data in the L2 cache.

Secondly, the circuitry of the L2 cache may be slower than the circuitry of the L1 cache in order to make the L2 cache less expensive. Third, L2 caches are usually 16 to 64 times larger than L1 caches, and larger memory subsystems tend to be slower than smaller ones. All this amounts to additional wait states for accessing data in the L2 cache. As noted earlier, the L2 cache can be as much as one order of magnitude slower than the L1 cache. The same situation occurs when you have to access data in the L3 cache.

The L1, L2, and L3 caches also differ in the amount of data the system fetches when there is a cache miss (see Chapter 6). When the CPU fetches data from or writes data to the L1 cache, it generally fetches or writes only the data requested. If you execute a `mov(al, memory);` instruction, the CPU writes only a single byte to the cache. Likewise, if you execute the `mov(mem32, eax);` instruction, the CPU reads exactly 32 bits from the L1 cache. However, access to memory subsystems below the L1 cache does not work in small chunks like this. Usually, memory subsystems move blocks of data, or *cache lines*, whenever accessing lower levels of the memory hierarchy. For example, if you execute the `mov(mem32, eax);` instruction, and `mem32`'s value is not in the L1 cache, the cache controller doesn't simply read `mem32`'s 32 bits from the L2 cache, assuming that it's present there. Instead, the cache controller will actually read a whole block of bytes (generally 16, 32, or 64 bytes, depending on the particular processor) from the L2 cache. The hope is that the program exhibits spatial locality so that reading a block of bytes will speed up future accesses to adjacent objects in memory. Unfortunately, the `mov(mem32, eax);` instruction doesn't complete until the L1 cache reads the entire cache line from the L2 cache. This excess time is known as *latency*. If the program does not access memory objects adjacent to `mem32` in the future, this latency is lost time.

A similar performance gulf separates the L2 and L3 caches and L3 and main memory. Main memory is typically one order of magnitude slower than the L3 cache; L3 accesses are much slower than L2 access. To speed up access to adjacent memory objects, the L3 cache reads data from main memory in cache lines. Likewise, L2 cache reads cache lines from L3.

Standard DRAM is two to three orders of magnitude faster than SSD storage (which is an order of magnitude faster than hard drives, which is why hard disks often have their own DRAM-based caches). To overcome this, there's usually a difference of two to three orders of magnitude in size between the L3 cache and the main memory so that the difference in speed between disk and main memory matches that between the main memory and the L3 cache. (Balancing performance characteristics in the memory hierarchy is a goal to strive for in order to effectively use the different types of memory.)

We won't consider the performance of the other memory-hierarchy subsystems in this chapter, as they are more or less under programmer control. Because their access is not automatic, very little can be said about how frequently a program will access them. However, in Chapter 12 we'll look at some considerations for these storage devices.

11.4 Cache Architecture

Up to this point, we have treated the cache as a magical place that automatically stores data when we need it, perhaps fetching new data as the CPU requires it. But how exactly does the cache do this? And what happens when it is full and the CPU is requesting additional data that's not there? In this section, we'll look at the internal cache organization and try to answer these two questions, along with a few others.

Programs access only a small amount of data at a given time, and a cache that is sized accordingly will improve their performance. Unfortunately, the data that programs want rarely sits in contiguous memory locations—it's usually spread out all over the address space. Therefore, cache design has to account for the fact that the cache must map data objects at widely varying addresses in memory.

As noted in the previous section, cache memory is not organized in a single group of bytes. Instead, it's usually organized in blocks of *cache lines*, with each line containing some number of bytes (typically a small power of 2 like 16, 32, or 64), as shown in Figure 11-2.

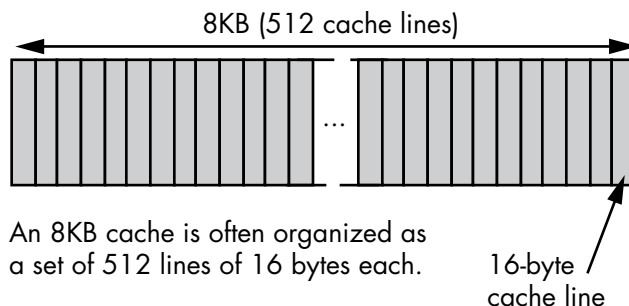


Figure 11-2: Possible organization of an 8KB cache

We can attach a different noncontiguous address to each of the cache lines. Cache line 0 might correspond to addresses \$10000 through \$1000F, and cache line 1 might correspond to addresses \$21400 through \$2140F. Generally, if a cache line is n bytes long, it will hold n bytes from main memory that fall on an n -byte boundary. In the example in Figure 11-2, the cache lines are 16 bytes long, so a cache line holds blocks of 16 bytes whose addresses fall on 16-byte boundaries in main memory (in other words, the LO 4 bits of the address of the first byte in the cache line are always 0).

When the cache controller reads a cache line from a lower level in the memory hierarchy, where does the data go in the cache? The answer is determined by the caching scheme in use. There are three different caching schemes: direct-mapped cache, fully associative cache, and n -way set associative cache.

11.4.1 Direct-Mapped Cache

In a *direct-mapped cache* (also known as the *one-way set associative cache*), a particular block of main memory is always loaded into—mapped to—the exact

same cache line, determined by a small number of bits in the data block's memory address. Figure 11-3 shows how a cache controller could select the appropriate cache line for an 8KB cache with 512 16-byte cache lines and a 32-bit main-memory address.

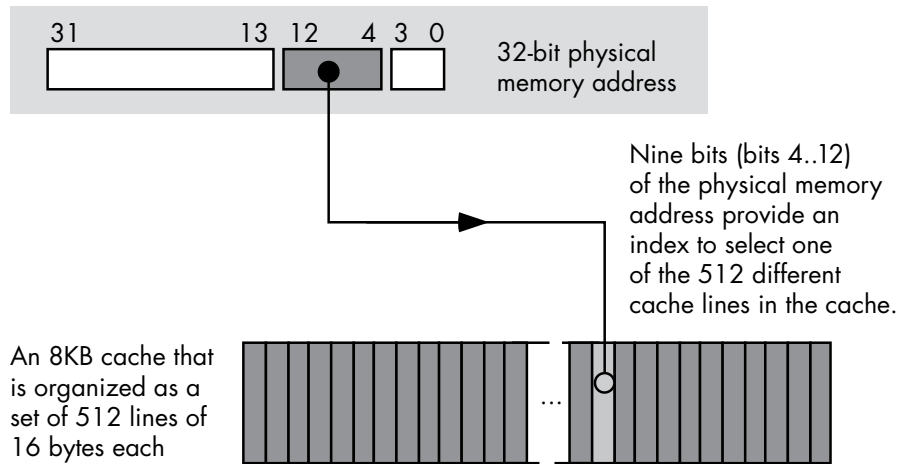


Figure 11-3: Selecting a cache line in a direct-mapped cache

A cache with 512 cache lines requires 9 bits to select one of the cache lines ($2^9 = 512$). In this example, bits 4 through 12 of the address determine which cache line to use (assuming we number the cache lines from 0 to 511), while bits 0 through 3 determine the particular byte within the 16-byte cache line.

The direct-mapped caching scheme is very easy to implement. Extracting 9 (or some other number of) bits from the memory address and using the result as an index into the array of cache lines is trivial and fast, though this design may not make effective use of all the cache memory.

For example, the caching scheme in Figure 11-3 maps address 0 to cache line 0. It also maps addresses \$2000 (8KB), \$4000 (16KB), \$6000 (24KB), \$8000 (32KB), and every other address that is a multiple of 8 kilobytes to cache line 0. This means that if a program is constantly accessing data at addresses that are multiples of 8KB and not accessing any other locations, the system will use only cache line 0, leaving all the other cache lines unused. In this extreme case, the cache is effectively limited to the size of one cache line, and each time the CPU requests data at an address that is mapped to, but not present in, cache line 0, it has to go down to a lower level in the memory hierarchy to access that data.

11.4.2 Fully Associative Cache

In a fully associative cache subsystem, the cache controller can place a block of bytes in any one of the cache lines present in the cache memory. While this is the most flexible cache system, the extra circuitry to achieve full associativity is expensive and, worse, can slow down the memory subsystem. Most L1 and L2 caches are not fully associative for this reason.

11.4.3 *n*-Way Set Associative Cache

If a fully associative cache is too complex, too slow, and too expensive to implement, but a direct-mapped cache is too inefficient, an *n*-way set associative cache is a compromise between the two. In an *n*-way set associative cache, the cache is broken up into sets of *n* cache lines. The CPU determines the particular set to use based on some subset of the memory address bits, just as in the direct-mapping scheme, and the cache controller uses a fully associative mapping algorithm to determine which one of the *n* cache lines within the set to use.

For example, an 8KB two-way set associative cache subsystem with 16-byte cache lines organizes the cache into 256 cache-line sets with two cache lines each. Eight bits from the memory address determine which one of these 256 different sets will contain the data. Once the cache-line set is determined, the cache controller maps the block of bytes to one of the two cache lines within the set (see Figure 11-4). This means two different memory addresses located on 8KB boundaries (addresses having the same value in bits 4 through 11) can both appear simultaneously in the cache. However, a conflict will occur if you attempt to access a third memory location at an address that is an even multiple of 8KB.

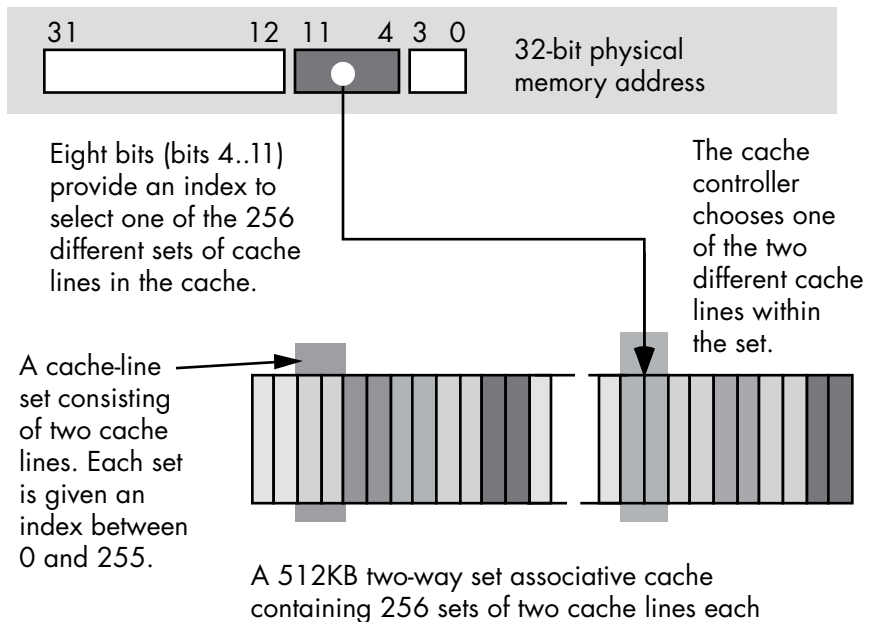


Figure 11-4: A two-way set associative cache

A four-way set associative cache puts four associative cache lines in each cache-line set. In an 8KB cache like the one in Figure 11-4, a four-way set associative caching scheme would have 128 cache-line sets with four cache lines each. This would allow the cache to maintain up to four different blocks of data without a conflict, each of which would map to the same cache line in a direct-mapped cache.

A two- or four-way set associative cache is much better than a direct-mapped cache and considerably less complex than a fully associative cache. The more cache lines we have in each cache-line set, the closer we come to creating a fully associative cache, with all the attendant problems of complexity and speed. Most cache designs are direct-mapped, two-way set associative, or four-way set associative. The various members of the 80x86 family make use of all three.

MATCHING THE CACHING SCHEME TO THE ACCESS TYPE

Despite its downsides, the direct-mapped cache is, in fact, very effective for data that you access sequentially rather than randomly. Because the CPU typically executes machine instructions sequentially, instruction bytes can be stored very effectively in a direct-mapped cache. However, programs tend to access data more randomly than they access code, so data is better stored in a two-way or four-way set associative cache.

Because of these different access patterns, many CPU designers use separate caches for data and machine instruction bytes—for example, an 8KB data cache and an 8KB instruction cache rather than a single 16KB unified cache. The advantage of this approach is that each cache can use the caching scheme that's most appropriate for the particular values it will store. The drawback is that the two caches are now each half the size of a unified cache, which may cause more cache misses than would occur with a unified cache. The choice of an appropriate cache organization is a difficult one, beyond the scope of this book, and can be made only after you've analyzed many programs running on the target processor.

11.4.4 Cache-Line Replacement Policies

Thus far, we've answered the question, "Where do we put a block of data in the cache?" Now we turn to the equally important question, "What happens if a cache line isn't available when we want to put a block of data in it?"

For a direct-mapped cache architecture, the cache controller simply replaces whatever data was formerly in the cache line with the new data. Any subsequent reference to the old data will result in a cache miss, and the cache controller will have to restore that old data to the cache by replacing whatever data is in that line.

For a two-way set associative cache, the replacement algorithm is a bit more complex. As you've seen, whenever the CPU references a memory location, the cache controller uses some subset of the address's bits to determine the cache-line set that should be used to store the data. Then, using some fancy circuitry, the cache controller determines whether the data is already present in one of the two cache lines in the destination set. If the data isn't there, the CPU has to retrieve it from memory, and the controller has to pick one of the two lines to use. If either or both of the cache lines

are currently unused, the controller picks an unused line. However, if both cache lines are currently in use, the controller must pick one of them and replace its data with the new data.

The controller cannot predict the cache line whose data will be referenced first and replace the other cache line, but it can use the principle of temporality: if a memory location has been referenced recently, it's likely to be referenced again in the very near future. This implies the following corollary: if a memory location hasn't been accessed in a while, it's likely to be a long time before the CPU accesses it again. Therefore, many cache controllers use the *least recently used (LRU)* algorithm.

An LRU policy is easy to implement in a two-way set associative cache system, using a single bit for each set of two cache lines. Whenever the CPU accesses one of the two cache lines this bit is set to 0, and whenever the CPU accesses the other cache line, this bit is set to 1. Then, when a replacement is necessary, the cache controller replaces the LRU cache line, indicated by the inverse of this bit.

For four-way (and greater) set associative caches, maintaining the LRU information is a bit more difficult, which is one reason the circuitry for such caches is more complex. Because of the complications LRU might introduce, other replacement policies are sometimes used instead. Two of them, *first-in, first-out (FIFO)* and *random*, are easier to implement than LRU, but they have their own problems. A full discussion of their pros and cons is beyond the scope of this book, but you can find more information in a text on computer architecture or operating systems.

11.4.5 Cache Write Policies

What happens when the CPU writes data to memory? The simple answer, and the one that results in the quickest operation, is that the CPU writes the data to the cache. However, what happens when the cache-line data is subsequently replaced by data that is read from memory? If the modified contents of the cache line are not written to main memory, they will be lost. The next time the CPU attempts to access that data, it will reload the cache line with the old data.

Clearly, any data written to the cache must ultimately be written to main memory as well. Caches use two common write policies: *write-through* and *write-back*.

The write-through policy states that any time data is written to the cache, the cache immediately turns around and writes a copy of that cache line to main memory. The CPU does not have to halt while the cache controller writes the data from cache to main memory. So, unless the CPU needs to access main memory shortly after the write occurs, this operation takes place in parallel with the program's execution. Because the write-through policy updates main memory with the new value as soon as possible, it is a better policy to use when two different CPUs are communicating through shared memory.

Still, a write operation takes some time, during which it's likely that a CPU will want to access main memory, so this policy may not be a

high-performance solution. Worse, suppose the CPU reads from and writes to the memory location several times in succession. With a write-through policy in place, the CPU will saturate the bus with cache-line writes, and this will significantly hamper the program's performance.

With the write-back policy, writes to the cache are not immediately written to main memory; instead, the cache controller updates main memory later. This scheme tends to be higher performance, because several writes to the same cache line within a short time period won't generate multiple writes to main memory.

To determine which cache lines must be written back to main memory, the cache controller usually maintains a *dirty bit* within each one. The cache system sets this bit whenever it writes data to the cache. At some later time, the cache controller checks the dirty bit to determine if it must write the cache line to memory. For example, whenever the cache controller replaces a cache line with other data from memory, it first checks the dirty bit, and if that bit is set, the controller writes that cache line to memory before going through with the cache-line replacement. Note that this increases the latency time during a cache-line replacement. This latency could be reduced if the cache controller were able to write dirty cache lines to main memory while no other bus access was occurring. Some systems provide this functionality, and others do not for economic reasons.

11.4.6 Cache Use and Software

A cache subsystem is not a panacea for slow memory access, and can in fact actually *hurt* an application's performance. In order for a cache system to be effective, software must be written with the cache behavior in mind. Particularly, good software must exhibit either spatial or temporal locality of reference—which the software designer accomplishes by placing oft-used variables adjacent in memory so they tend to fall into the same cache lines—and avoid data structures and access patterns that force the cache to frequently replace cache lines.

Suppose that an application accesses data at several different addresses that the cache controller would map to the same cache line. With each access, the cache controller must read in a new cache line (possibly flushing the old one back to memory if it is dirty). As a result, each memory access incurs the latency cost of retrieving a cache line from main memory. This degenerate case, known as *thrashing*, can slow down the program by one to two orders of magnitude, depending on the speed of main memory and the size of a cache line. We'll take another look at thrashing a little later in this chapter.

A benefit of the cache subsystem on modern 80x86 CPUs is that it automatically handles many misaligned data references. Remember, there's a performance penalty for accessing words or double-word objects at an address that is not an even multiple of that object's size. By providing some fancy logic, Intel's designers have eliminated this penalty as long as the data object is located completely within a cache line. However, if the object crosses a cache line, the penalty still applies.

11.5 NUMA and Peripheral Devices

Although most of the RAM in a system is based on high-speed DRAM interfaced directly with the processor's bus, not all memory is connected to the CPU this way. Sometimes a large block of RAM is part of a peripheral device—for example, a video card, network interface card, or USB controller—and you communicate with that device by writing data to its RAM. Unfortunately, the access time to the RAM on these peripheral devices is often much slower than the access time to main memory. In this section, we'll use the video card as an example, although NUMA performance applies to other devices and memory technologies as well.

A typical video card interfaces with a CPU through a *Peripheral Component Interconnect Express (PCI-e)* bus inside the computer system. Though 16-lane PCI-e buses are fast, memory access is still much faster. Game programmers long ago discovered that manipulating a copy of the screen data in main memory and writing that data to the video card RAM only periodically (typically once every 1/60 of a second during video retrace, to avoid flicker) is much faster than writing directly to the video card every time you want to make a change.

Caches and the virtual memory subsystem operate transparently (that is, applications are unaware of the underlying operations taking place), but NUMA memory does not, so programs that write to NUMA devices must minimize the number of accesses whenever possible (for example, by using an offscreen bitmap to hold temporary results). If you're actually storing and retrieving data on a NUMA device, like a flash memory card, you must explicitly cache the data yourself.

11.6 Virtual Memory, Memory Protection, and Paging

In a modern operating system such as Android, iOS, Linux, macOS, or Windows, it is very common to have several different programs running concurrently in memory. This presents several problems:

- How do you keep the programs from interfering with one another's memory?
- If two programs both expect to load a value into memory at address \$1000, how can you load both values and execute both programs at the same time?
- What happens if the computer has 64GB of memory, and you decide to load and execute three different applications, two of which require 32GB and one that requires 16GB (not to mention the memory that the OS requires for its own purposes)?

The answers to all these questions lie in the virtual memory subsystem that modern processors support.

Virtual memory on CPUs such as the 80x86 gives each process its own 32-bit address space.² This means that address \$1000 in one program is physically different from address \$1000 in a separate program. The CPU achieves this sleight of hand by mapping the *virtual addresses* used by programs to different *physical addresses* in actual memory. The virtual address and the physical address don't have to be the same, and usually they aren't. For example, program 1's virtual address \$1000 might actually correspond to physical address \$215000, while program 2's virtual address \$1000 might correspond to physical memory address \$300000. The CPU accomplishes this using *paging*.

The concept behind paging is quite simple. First, you break up memory into blocks of bytes called *pages*. A page in main memory is comparable to a cache line in a cache subsystem, although pages are usually much larger than cache lines. For example, the 32-bit 80x86 CPUs use a page size of 4,096 bytes; 64-bit variants allow larger page sizes.

For each page, you use a lookup table to map the HO bits of a virtual address to the HO bits of the physical address in memory, and you use the LO bits of the virtual address as an index into that page. For example, with a 4,096-byte page, you'd use the LO 12 bits of the virtual address as the offset (0.4095) within the page, and the upper 20 bits as an index into a lookup table that returns the actual upper 20 bits of the physical address (see Figure 11-5).

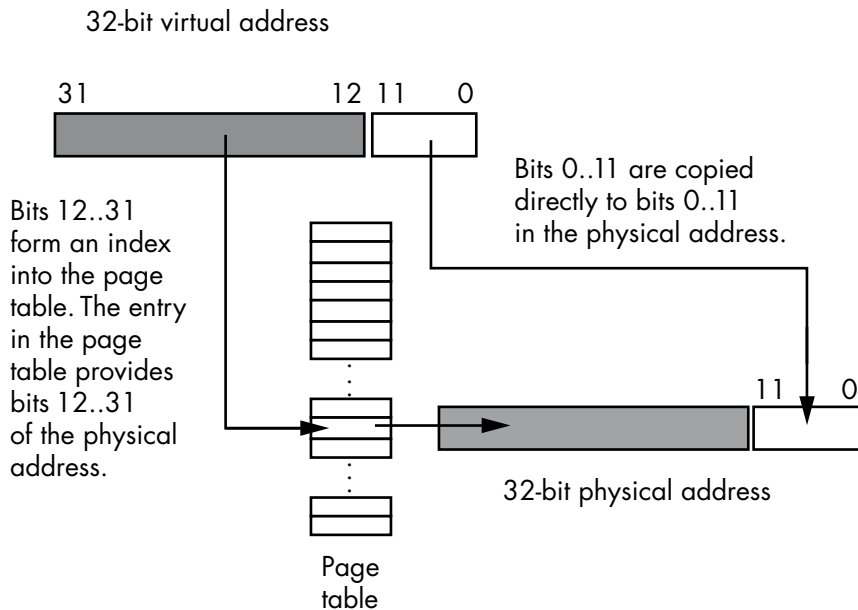


Figure 11-5: Translating a virtual address to a physical address

2. On newer 64-bit processors, of course, each process gets its own 64-bit address space.

A 20-bit index into the page table would require over one million entries in the page table. If each entry is a 32-bit value, the page table would be 4MB long—larger than many of the programs that would run in memory! However, by using a multilevel page table, you can easily create a page table for most small programs that is only 8KB long. The details are unimportant here. Just rest assured that you don't need a 4MB page table unless your program consumes the entire 4GB address space.

If you study Figure 11-5 for a few moments, you'll probably discover one problem with using a page table—it requires two separate memory accesses in order to retrieve the data stored at a single physical address in memory: one to fetch a value from the page table, and one to read from or write to the desired memory location. To prevent cluttering the data or instruction cache with page-table entries, which increases the number of cache misses for data and instruction requests, the page table uses its own cache, known as the *translation lookaside buffer (TLB)*. This cache typically has 64 to 512 entries on modern Intel processors—enough to handle a fair amount of memory without a miss. Because a program typically works with less data than this at any given time, most page-table accesses come from the cache rather than main memory.

As noted, each entry in the page table contains 32 bits, even though the system really only needs 20 bits to remap each virtual address to a physical address. Intel, on the 80x86, uses some of the remaining 12 bits to provide memory protection information:

- One bit marks whether a page is read/write or read-only.
- One bit determines whether you can execute code on that page.
- A number of bits determine whether the application can access that page or if only the operating system can do so.
- A number of bits determine if the CPU has written to the page but hasn't yet written to the physical memory address corresponding to it (that is, whether the page is “dirty” or not, and whether the CPU has accessed the page recently).
- One bit determines whether the page is actually present in physical memory or if it exists on secondary storage somewhere.

Your applications do not have access to the page table (reading and writing the page table is the operating system's responsibility), so they cannot modify these bits. However, some operating systems provide functions you can call if you want to change certain bits in the page table (for example, Windows allows you to set a page to read-only).

Beyond remapping memory so multiple programs can coexist in main memory, paging also provides a mechanism whereby the operating system can move infrequently used pages to secondary storage. Locality of reference applies not only to cache lines but to pages in main memory as well. At any given time, a program will access only a small percentage of the pages in main memory that contain data and instruction bytes; this set of pages is known as the *working set*. Although the working set varies slowly over time,

for small periods of time it remains constant. Therefore, there's little need for the remainder of the program to consume valuable main-memory storage that some other process could be using. If the operating system can save the currently unused pages to disk, the main memory they would consume is available for other programs that need it.

Of course, the problem with moving data out of main memory is that eventually the program might actually need it. If you attempt to access a page of memory, and the page-table bit tells the memory management unit (MMU) that the page isn't present in main memory, the CPU interrupts the program and passes control to the operating system. The operating system reads the corresponding page of data from the disk drive and copies it to some available page in main memory. This process is nearly identical to the process used by a fully associative cache subsystem, except that accessing the disk is much slower than accessing main memory. In fact, you can think of main memory as a fully associative write-back cache with 4,096-byte cache lines, which caches the data that is stored on the disk drive. Placement and replacement policies and other behaviors are very similar for caches and main memory.

NOTE

For more information on how the operating system swaps pages between main memory and secondary storage, consult a textbook on operating system design.

Because each program has a separate page table, and programs themselves don't have access to the page tables, programs cannot interfere with one another's operation. That is, a program cannot change its page tables in order to access data found in another process's *address space*. If your program crashes by overwriting itself, it cannot crash other programs at the same time. This is a big benefit of a paging memory system.

If two programs want to cooperate and share data, they can do so by placing that data in a memory area that they share. All they have to do is tell the operating system that they want to share some pages of memory. The operating system returns to each process a pointer to some segment of memory whose physical address is the same for both processes. Under Windows, you can achieve this by using *memory-mapped files*; see the operating system documentation for more details. macOS and Linux also support memory-mapped files as well as some special shared-memory operations; again, see the OS documentation for more details.

Although this discussion applies specifically to the 80x86 CPU, multi-level paging systems are common on other CPUs as well. Page sizes tend to vary from about 1KB to 4MB, depending on the CPU. For CPUs that support an address space larger than 4GB, some CPUs use an *inverted page table* or a *three-level page table*. Although the details are beyond the scope of this chapter, the basic principle remains the same: the CPU moves data between main memory and the disk in order to keep oft-accessed data in main memory as much of the time as possible. These other page-table schemes are good at reducing the size of the page table when an application uses only a fraction of the available memory space.

THRASHING

Thrashing is a degenerate case that can cause the overall system performance to drop to the speed of a lower level in the memory hierarchy, like main memory or, worse yet, the disk drive. There are two primary causes of thrashing:

- Insufficient memory at a given level in the memory hierarchy to properly contain the programs' working sets of cache lines or pages
- A program that does not exhibit locality of reference

If there is insufficient memory to hold a working set of pages or cache lines, the memory system will constantly be replacing one block of data in the cache or main memory with another block of data from main memory or the disk. As a result, the system winds up operating at the speed of the slower memory in the memory hierarchy. A common example of thrashing occurs with virtual memory. A user may have several applications running at the same time, and the sum total of the memory required by these programs' working sets is greater than all of the physical memory available to the programs. As a result, when the operating system switches between the applications it has to copy each application's data, and possibly program instructions, to and from disk. Because switching between programs is often much faster than retrieving data from the disk, this slows the programs down tremendously.

As already discussed, if the program does not exhibit locality of reference and the lower memory subsystems are not fully associative, thrashing can occur even if there is free memory at the current level in the memory hierarchy. To revisit our earlier example, suppose an 8KB L1 caching system uses a direct-mapped cache with 512 16-byte cache lines. If a program references data objects 8KB apart on every access, the system will have to replace the same line in the cache over and over again with the data from main memory. This occurs even though the other 511 cache lines are currently unused.

To reduce thrashing when insufficient memory is the problem, you can simply add memory. If that's not an option, you can try to run fewer processes concurrently or modify your program so that it references less memory over a given period. To reduce thrashing when locality of reference is the culprit, you should restructure your program and its data structures so its memory references are physically closer.

11.7 Writing Software That Is Cognizant of the Memory Hierarchy

Software that is aware of memory performance behavior can run much faster than software that is not. Although a system's caching and paging facilities may perform reasonably well for typical programs, it's easy to write software that would run faster even if the caching system were not present. The best software is written to take maximum advantage of the memory hierarchy.

A classic example of a bad design is the following loop, which initializes a two-dimensional array of integer values:

```
int array[256][256];  
    . . .  
    for( i=0; i<256; ++i )  
        for( j=0; j<256; ++j )  
            array[j][i] = i*j;
```

Believe it or not, that code runs much slower on a modern CPU than the following sequence:

```
int array[256][256];  
    . . .  
    for( i=0; i<256; ++i )  
        for( j=0; j<256; ++j )  
            array[i][j] = i*j;
```

The only difference between the two code sequences is that the *i* and *j* indices are swapped when accessing elements of the array. This minor modification can be responsible for a one or two order of magnitude difference in their respective runtimes! To understand why, remember that the C programming language uses row-major ordering for two-dimensional arrays in memory. That means the second code sequence accesses sequential locations in memory, exhibiting spatial locality of reference. The first code sequence, however, accesses array elements in the following order:

```
array[0][0]  
array[1][0]  
array[2][0]  
array[3][0]  
    . . .  
array[254][0]  
array[255][0]  
array[0][1]  
array[1][1]  
array[2][1]  
    . . .
```

If integers are 4 bytes each, then this sequence will access the double-word values at offsets 0; 1,024; 2,048; 3,072; and so on, from the base address of the array, which are distinctly *not* sequential. Most likely, this code will load only *n* integers into an *n*-way set associative cache and then immediately cause thrashing thereafter, as each subsequent array element has to be copied from the cache into main memory to prevent that data from being overwritten.

The second code sequence does not exhibit thrashing. Assuming 64-byte cache lines, the second code sequence will store 16 integer values into the same cache line before having to load another cache line from main memory, replacing an existing one. As a result, this second code sequence spreads out the cost of retrieving the cache line from memory

over 16 memory accesses rather than over a single access, as occurs with the first code sequence.

In addition to accessing variables sequentially in memory, there are several other variable declaration tricks you can use to maximize the performance of the memory hierarchy. First, declare together all variables you use within a common code sequence. In most languages, this will allocate storage for the variables in physically adjacent memory locations, thus supporting spatial locality as well as temporal locality. Second, use local (automatic) variables, because most languages allocate local storage on the stack and, as the system references the stack frequently, variables on the stack tend to be in the cache. Third, declare your scalar variables together, and separately from your array and record variables. Access to any one of several adjacent scalar variables generally forces the system to load all of the adjacent objects into the cache.

In general, study the memory access patterns your program exhibits and adjust your application accordingly. You can spend hours rewriting your code in hand-optimized assembly language trying to achieve a 10 percent performance improvement, but if you instead modify the way your program accesses memory, it's not unheard of to see an order of magnitude improvement in performance.

11.8 Runtime Memory Organization

Operating systems like macOS, Linux, or Windows put different types of data into different sections (or *segments*) of main memory. Although it's possible to control the memory organization by running a linker and specifying various parameters, by default Windows loads a typical program into memory using the organization shown in Figure 11-6 (macOS and Linux are similar, though they rearrange some of the sections).

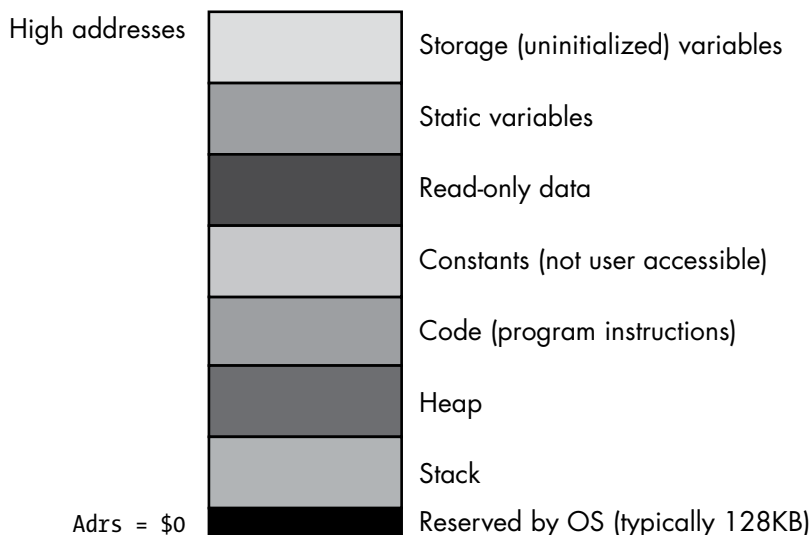


Figure 11-6: Typical Windows runtime memory organization

The operating system reserves the lowest memory addresses, and your application generally cannot access data (or execute instructions) at these addresses. One reason the OS reserves this space is to help detect NULL pointer references. Programmers often initialize a pointer with NULL (0) to indicate that it is not valid. Should you attempt to access memory location 0 under such an OS, it will generate a *general protection fault* to indicate that you've accessed a memory location that doesn't contain valid data.

The remaining seven sections of memory hold different types of data associated with your program:

- The code section holds the program's machine instructions.
- The constant section contains compiler-generated read-only data.
- The read-only data section holds user-defined data that can only be read, never written.
- The static section stores user-defined, initialized, static variables.
- The storage, or BSS, section holds user-defined uninitialized variables.
- The stack section maintains local variables and other temporary data.
- The heap section maintains dynamic variables.

NOTE

Often, a compiler will combine the code, constant, and read-only data sections because they all contain read-only data.

Most of the time, a given application can live with the default layouts chosen for these sections by the compiler and linker/loader. In some cases, however, knowing the memory layout can help you develop shorter programs. For example, combining the code, constants, and read-only data sections into a single read-only section can save padding space that the compiler/linker might otherwise place between them. Although these savings are probably insignificant for large applications, they can have a big impact on the size of a small program.

The following sections discuss each of these memory areas in detail.

11.8.1 Static and Dynamic Objects, Binding, and Lifetime

Before exploring the memory organization of a typical program, we need to define a few terms: binding, lifetime, static, and dynamic.

Binding is the process of associating an attribute with an object. For example, when you assign a value to a variable, the value is *bound* to that variable at the point of the assignment. This bond remains until you bind some other value to the variable (via another assignment operation). Likewise, if you allocate memory for a variable while the program is running, the variable is bound to the address at that point. They remain bound until you associate a different address with the variable. Binding needn't occur at runtime. For example, values are bound to constant objects during compilation, and these bonds cannot change while the program is running.

The *lifetime* of an attribute extends from the point when you first bind that attribute to an object to the point when you break that bond, perhaps

by binding a different attribute to the object. For example, the lifetime of a variable is from the time you first associate memory with the variable to the moment you deallocate that variable's storage.

Static objects are those that have an attribute bound to them prior to the application's execution (usually during compilation or during the linking phase, though it is possible to bind values even earlier). Constants are good examples of static objects; they have the same value bound to them throughout program execution. Global (program-level) variables in programming languages like Pascal, C/C++, and Ada are also examples of static objects in that they have the same address bound to them throughout the program's lifetime. The lifetime of a static object, therefore, extends from the point at which the program first begins execution to the point when the application terminates.

Associated with static binding is the notion of identifier *scope*—the section of the program where the identifier's name is bound to the object. As names exist only during compilation, scope qualifies as a static attribute in compiled languages. (In interpretive languages, where the interpreter maintains the identifier names during program execution, scope can be a nonstatic attribute.) The scope of a local variable is generally limited to the procedure or function in which you declare it (or to any nested procedure or function declarations in block structured languages like Pascal or Ada), and the name is not visible outside the subroutine. In fact, it's possible to reuse an identifier's name in a different scope (that is, in a different function or procedure). In that case, the second occurrence of the identifier will be bound to a different object than its first occurrence.

Dynamic objects are those that have some attribute assigned to them during program execution. While it is running, the program may choose to change that attribute (*dynamically*). The lifetime of that attribute begins when the application binds the attribute to the object and ends when the program breaks that bond. If the program never breaks the bond, the attribute's lifetime extends from the point of association to the point the program terminates. The system binds dynamic attributes to an object at runtime, after the application begins execution.

NOTE

An object may have a combination of static and dynamic attributes. For example, a static variable has an address bound to it for the entire execution time of the program, but it could have different values bound to it throughout the program's lifetime. Any given attribute, however, is either static or dynamic; it cannot be both.

11.8.2 The Code, Read-Only, and Constant Sections

The code section in memory contains the machine instructions for a program. Your compiler translates each statement you write into a sequence of one or more byte values. The CPU interprets these byte values as machine instructions during program execution.

Most compilers also attach a program's read-only data to the code section because, like the code instructions, the read-only data is already write-protected. However, it is perfectly possible under Windows, macOS,

Linux, and many other operating systems to create a separate section in the executable file and mark it as read-only. As a result, some compilers support a separate read-only data section. Such sections contain initialized data, tables, and other objects that the program should not change during program execution.

The constant section shown in Figure 11-6 typically contains data that the compiler generates (as opposed to user-defined read-only data). Most compilers actually emit this data directly to the code section. This is why, as previously noted, in most executable files, you'll find a single section that combines the code, read-only data, and constant data sections.

11.8.3 The Static Variables Section

Many languages enable you to initialize a global variable during the compilation phase. For example, in C/C++ you could use statements like the following to provide initial values for these static objects:

```
static int i = 10;
static char ch[] = { 'a', 'b', 'c', 'd' };
```

In C/C++ and other languages, the compiler places these initial values in the executable file. When you execute the application, the OS loads the portion of the executable file that contains these static variables into memory so that the values appear at the addresses associated with those static variables. Therefore, when the program shown here first begins execution, *i* and *ch* will have these values bound to them.

11.8.4 The Storage Variables Section

The storage variables (or BSS) section is where compilers typically put static objects that don't have an explicit value associated with them. BSS stands for "block started by a symbol," which is an old assembly language term describing a pseudo-opcode you would use to allocate storage for an uninitialized static array. In modern operating systems like Windows and Linux, the compiler/linker puts all uninitialized variables into a BSS section that simply tells the OS how many bytes to set aside for that section. When the OS loads the program into memory, it reserves sufficient memory for all the objects in the BSS section and fills this range of memory with *os*.

Note that the BSS section in the executable file doesn't actually contain any data, so programs that declare uninitialized static objects (especially large arrays) in a BSS section will consume less disk space.

However, not all compilers actually use a BSS section. Some Microsoft languages and linkers, for example, simply place the uninitialized objects in the static/read-only data section and explicitly give them an initial value of 0. Although Microsoft claims that this scheme is faster, it certainly makes executable files larger if your code has large, uninitialized arrays (because each byte of the array winds up in the executable file—something that would not happen if the compiler placed the array in a BSS section).

11.8.5 The Stack Section

The stack is a data structure that expands and contracts in response to procedure invocations and returns to calling routines, among other things. At runtime, the system places all automatic variables (nonstatic local variables), subroutine parameters, temporary values, and other objects in the stack section of memory in a special data structure called an *activation record* (which is aptly named, as the system creates it when a subroutine first begins execution, and deallocates it when the subroutine returns to its caller). Therefore, the stack section in memory is very busy.

Most CPUs implement the stack using a register called the *stack pointer*. Some CPUs, however, don't provide an explicit stack pointer, instead using a general-purpose register for stack implementation. If a CPU provides a stack pointer, we say that it supports a *hardware stack*; if it uses a general-purpose register, then we say that it uses a *software-implemented stack*. The 80x86 provides a hardware stack, while the MIPS Rx000 CPU family uses a software-implemented stack. Systems that provide hardware stacks can generally manipulate data on the stack using fewer instructions than systems that implement the stack in software. In theory, a hardware stack actually slows down all instructions the CPU executes, but in practice, the 80x86 CPU is one of the fastest CPUs around, providing ample proof that having a hardware stack doesn't necessarily mean you'll wind up with a slow CPU.

11.8.6 The Heap Section and Dynamic Memory Allocation

Although simple programs may need only static and automatic variables, sophisticated programs need to be able to allocate and deallocate storage dynamically (at runtime) under program control. The C and HLA languages provide the `malloc()` and `free()` functions for this purpose, C++ provides `new()` and `delete()`, Pascal uses `new()` and `dispose()`, and other languages include comparable routines. These memory allocation routines have a few things in common: they let the programmer request how many bytes of storage to allocate, they return a *pointer* to the newly allocated storage (that is, the address of that storage), and they provide a facility for returning the storage space to the system once it is no longer needed, so the system can reuse it in a future allocation call. Dynamic memory allocation takes place in a section of memory known as the *heap*.

Generally, an application refers to data on the heap using pointer variables either implicitly or explicitly; some languages, like Java, implicitly use pointers behind the scenes. Thus, objects in heap memory are usually known as *anonymous variables* because we refer to them by their memory address (via pointers) rather than by name.

The OS and application create the heap section in memory after the program begins execution; the heap is never a part of the executable file. Generally, the OS and language runtime libraries maintain the heap for an application. Despite the variations in memory management implementations, it's still a good idea for you to have a basic idea of how heap allocation and deallocation operate, because using them inappropriately will have a very negative impact on your application performance.

11.8.6.1 A Simple Memory Allocation Scheme

An extremely simple (and fast) memory allocation scheme would return a pointer to a block of memory whose size the caller requests. It would carve out allocation requests from the heap, returning blocks of memory that are currently unused.

A very simple memory manager might maintain a single variable (a *free space* pointer) pointing to the heap. Whenever a memory allocation request comes along, the system makes a copy of this heap pointer and returns it to the application; then the heap management routines add the size of the memory request to the address held in the pointer variable and verify that the memory request doesn't try to use more memory than is available in the heap (some memory managers return an error indication, like a NULL pointer, when the memory request is too large, and others raise an exception). As the heap management routines increment the free space pointer, they effectively mark all previous memory as "unavailable for future requests."

11.8.6.2 Garbage Collection

The problem with this simple memory management scheme is that it wastes memory, because there's no *garbage collection* mechanism for the application to free the memory so it can be reused later. Garbage collection—that is, reclaiming memory when an application has finished using it—is one of the main purposes of a heap management system.

The only catch is that supporting garbage collection requires some overhead. The memory management code will need to be more sophisticated, will take longer to execute, and will require some additional memory to maintain the internal data structures the heap management system uses.

Let's consider an easy implementation of a heap manager that supports garbage collection. This simple system maintains a (linked) list of free memory blocks. Each free memory block in the list requires two double-word values: one specifying the size of the free block, and the other containing a link to the next free block in the list (that is, a pointer), as shown in Figure 11-7.

The system initializes the heap with a NULL link pointer, and the size field contains the size of the heap's entire free space. When a memory allocation request comes along, the heap manager searches through the list to find a free block with enough memory to satisfy the request. This search process is one of the defining characteristics of a heap manager. Some common search algorithms are first-fit search and best-fit search. A *first-fit search*, as its name suggests, scans the list of blocks until it finds the *first* block of memory large enough to satisfy the allocation request. A *best-fit search* scans the entire list and finds the *smallest* block large enough to satisfy the request. The advantage of the best-fit algorithm is that it tends to preserve larger blocks better than the first-fit algorithm, so the system is still able to satisfy larger subsequent allocation requests when they arrive. The first-fit algorithm, on the other hand, just grabs the first suitably large block it finds, even if there's a smaller block that would suffice, which may limit the system's ability to handle future large memory requests.

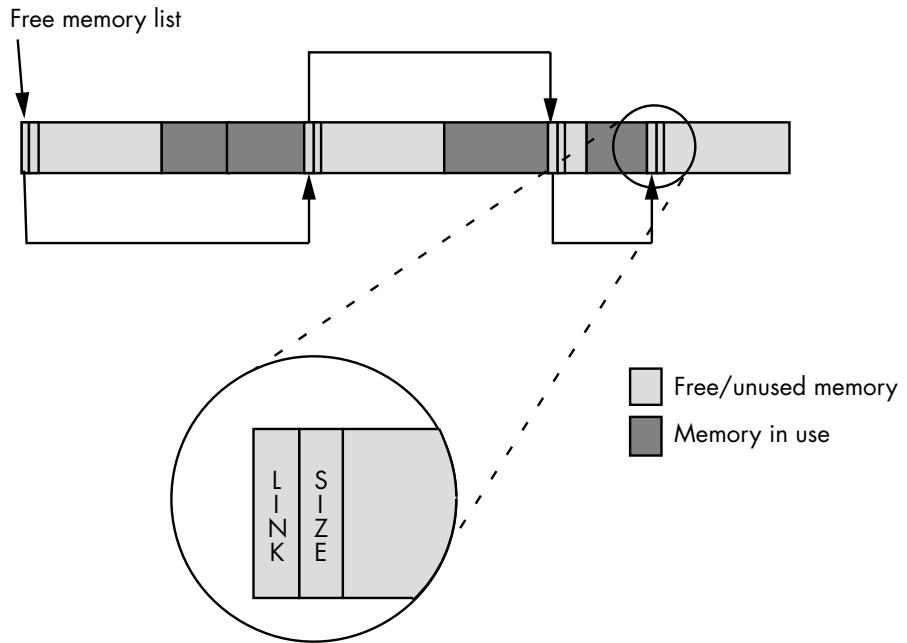


Figure 11-7: Heap management using a list of free memory blocks

Still, the first-fit algorithm does have a couple of advantages over the best-fit algorithm. The most obvious is that it is usually faster. The best-fit algorithm has to scan through every block in the free block list in order to find the smallest one large enough to satisfy the allocation request (unless, of course, it finds a perfectly sized block along the way). The first-fit algorithm can stop once it finds a block large enough to satisfy the request.

The first-fit algorithm also tends to suffer less from a degenerate condition known as *external fragmentation*. Fragmentation occurs after a long sequence of allocation and deallocation requests. Remember, when the heap manager satisfies a memory allocation request, it usually creates two blocks of memory: one in-use block for the request, and one free block that contains the remaining bytes from the original block (assuming the request did not exactly match the block size). After operating for a while, the best-fit algorithm may have produced lots of leftover blocks of memory that are too small to satisfy an average memory request, making them effectively unusable. As these small fragments accumulate throughout the heap, they can end up consuming a fair amount of memory. This can lead to a situation where the heap doesn't have a sufficiently large block to satisfy a memory allocation request even though there is enough total free memory available (spread throughout the heap). See Figure 11-8 for an example of this condition.

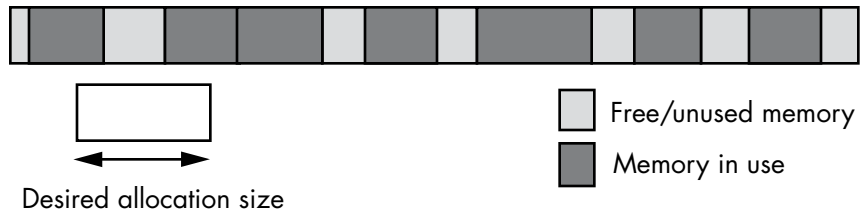


Figure 11-8: Memory fragmentation

There are other memory allocation strategies in addition to the first-fit and best-fit search algorithms. Some of these execute faster, some have less memory overhead, some are easy to understand (and some are very complex), some produce less fragmentation, and some can combine and use noncontiguous blocks of free memory. Memory/heap management is one of the more heavily studied subjects in computer science, and there's a considerable amount of literature explaining the benefits of one scheme over another. For more information on memory allocation strategies, check out a good book on OS design.

11.8.6.3 Freeing Allocated Memory

Memory allocation is only half of the story. As mentioned earlier, the heap manager has to provide a call that allows an application to return memory it no longer needs for future reuse. In C and HLA, for example, an application accomplishes this by calling the `free()` function.

At first blush, `free()` might seem like a very simple function to write: just append the previously allocated and now unused block to the end of the free list. The problem with this trivial implementation is that it almost guarantees that the heap becomes fragmented to the point of being unusable in very short order. Consider the situation in Figure 11-9.

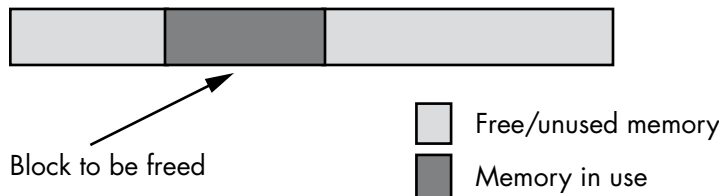


Figure 11-9: Freeing a memory block

If a trivial implementation of `free()` simply takes the block to be freed and appends it to the free list, the memory organization in Figure 11-9 produces three free blocks. However, because these three blocks are contiguous, the heap manager should really combine them into a single free block, so that it will be able to satisfy a larger request. Unfortunately, this operation would require it to scan the free block list to determine if there are any free blocks adjacent to the block the system is freeing.

While you could come up with a data structure that makes it easier to combine adjacent free blocks, such schemes generally add 8 or more bytes of overhead with each block on the heap. Whether or not this is a reasonable tradeoff depends on the average size of a memory allocation. If the applications that use the heap manager tend to allocate small objects, the extra overhead for each memory block could wind up consuming a large percentage of the heap space. However, if the most allocations are large, then the few bytes of overhead won't matter much.

11.8.6.4 The OS and Memory Allocation

The performance of the algorithms and data structures used by the heap manager is only one piece of the performance puzzle. Ultimately, the heap manager needs to request blocks of memory from the operating system. At one extreme, the OS handles all memory allocation requests directly. At the other extreme, the heap manager is a runtime library routine that links with your application, first requesting large blocks of memory from the OS and then doling out pieces of them as allocation requests arrive from the application.

The problem with making direct memory allocation requests to the operating system is that OS API calls are often very slow. This is because they generally involve switching between kernel mode and user mode on the CPU (which is not fast). Therefore, a heap manager that the OS implements directly will not perform well if your application makes frequent calls to the memory allocation and deallocation routines.

Because of the high overhead of an OS call, most languages implement their own versions of the `malloc()` and `free()` functions within their runtime library. On the very first memory allocation, the `malloc()` routine requests a large block of memory from the OS, and the application's `malloc()` and `free()` routines manage this block of memory themselves. If an allocation request comes along that the `malloc()` function cannot fulfill in the block it originally created, `malloc()` will request another large block (generally much larger than the request) from the OS and add that block to the end of its free list. Because the application's `malloc()` and `free()` routines call the OS only occasionally, the application doesn't suffer the performance hit associated with frequent OS calls.

Most standard heap management functions perform reasonably for a typical program. However, keep in mind that the procedures are very implementation- and language-specific; it's dangerous to assume that `malloc()` and `free()` are relatively efficient when writing software that requires high-performance components. The only portable way to ensure a high-performance heap manager is to develop your own application-specific set of allocation/deallocation routines. Writing such routines is beyond the scope of this book, but you should know you have this option.

11.8.6.5 Heap Memory Overhead

A heap manager often exhibits two types of overhead: performance (speed) and memory (space). Until now, this discussion has mainly dealt with the performance aspects, but now we'll turn our attention to memory.

Each block the system allocates requires some amount of overhead beyond the storage the application requests; at the very least, this overhead is a few bytes to keep track of the block's size. Fancier (higher-performance) schemes may require additional bytes, but typically the overhead is between 4 and 16 bytes. The heap manager can keep this information in a separate internal table, or it can attach the block size and other memory management information directly to the block it allocates.

Saving this information in an internal table has a couple of advantages. First, it is difficult for the application to accidentally overwrite the information stored there; attaching the data to the heap memory blocks themselves doesn't protect as well against this possibility. Second, putting memory management information in an internal data structure allows the memory manager to determine whether a given pointer is valid (that is, whether it points at some block of memory that the heap manager believes it has allocated).

The advantage of attaching the control information directly to each block that the heap manager allocates is that it's very easy to locate this information, whereas storing the information in an internal table might require a search operation.

Another issue that affects the overhead associated with the heap manager is the *allocation granularity*—the minimum number of bytes the heap manager supports. Although most heap managers allow you to request an allocation as small as 1 byte, they may actually allocate some minimum number of bytes greater than 1. To ensure an allocated object is aligned on a reasonable address for that object, most heap managers allocate memory blocks on a 4-, 8-, or 16-byte boundary. For performance reasons, many heap managers begin each allocation on a typical cache-line boundary, usually 16, 32, or 64 bytes.

Whatever the granularity, if the application requests some number of bytes that is less than or not a multiple of the heap manager's granularity, the heap manager will allocate extra bytes of storage so that the complete allocation is an even multiple of the granularity value. This amount varies by heap manager (and possibly even by version of a specific heap manager), so an application should never assume that it has more memory available than it requests.

The extra memory the heap manager allocates results in another form of fragmentation called *internal fragmentation*. Like external fragmentation, internal fragmentation produces small amounts of leftover memory throughout the system that cannot satisfy future allocation requests. Assuming random-sized memory allocations, the average amount of internal fragmentation that occurs on each allocation is half the granularity size. Fortunately, the granularity size is quite small for most memory managers (typically 16 bytes or less), so after thousands and thousands of memory allocations you'll lose only a couple dozen or so kilobytes to internal fragmentation.

Between the costs associated with allocation granularity and the memory control information, a typical memory request may require between 4 and 16 bytes, plus whatever the application requests. If you're making large memory allocation requests (hundreds or thousands of bytes), the overhead bytes won't consume a large percentage of memory on the heap. However, if you allocate lots of small objects, the memory consumed by internal fragmentation and memory control information may represent a significant portion of your heap area. For example, consider a simple memory manager that always allocates blocks of data on 4-byte boundaries and requires a single 4-byte length value that it attaches to each allocation request for memory storage. This means that the minimum amount of storage the heap manager requires for each allocation is 8 bytes. If you make a series of `malloc()` calls to allocate a single byte, the application won't be able to use almost 88 percent of the memory it allocates. Even if you allocate 4-byte values on each allocation request, the heap manager consumes 67 percent of the memory for overhead purposes. However, if your average allocation is a block of 256 bytes, the overhead requires only about 2 percent of the total memory allocation. In short, the larger your allocation request, the less impact the control information and internal fragmentation will have on your heap.

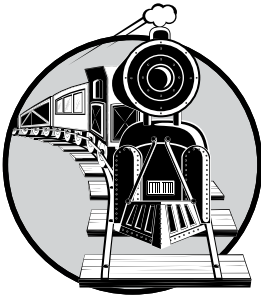
Many software engineering studies in computer science journals have found that memory allocation/deallocation requests cause a significant loss of performance. In such studies, the authors often obtained performance improvements of 100 percent or better just by implementing their own simplified, application-specific, memory management algorithms rather than calling the standard runtime library or OS kernel memory allocation code. Hopefully, this section has made you aware of this potential problem in your own code.

11.9 For More Information

Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Elsevier, 2012.

12

INPUT AND OUTPUT



A typical program has three basic tasks: input, computation, and output. So far we've concentrated on the computational aspects of the computer system, but now we'll turn to input and output.

This chapter will focus on the primitive input/output (I/O) activities of the CPU, rather than on the abstract file or character I/O that high-level applications usually employ. It will discuss how the CPU transfers data to and from the outside world, paying special attention to the performance issues behind I/O operations. As all high-level I/O activities are eventually routed through the low-level I/O systems, it's crucial to understand how these processes work if you want to write programs that communicate efficiently with the outside world.

12.1 Connecting a CPU to the Outside World

The first thing to know is that I/O in a typical computer system is radically different from I/O in a typical high-level programming language. At the primitive I/O levels of a computer system, you'll rarely find machine instructions that behave like Pascal's `writeln`, C++'s `cout`, C's `printf`, Swift's `print`, or even the HLA `stdin` and `stdout` statements. In fact, most I/O machine instructions behave exactly like the 80x86's `mov` instruction. To send data to an output device, the CPU simply moves that data to a special memory location; and to read data from an input device, the CPU retrieves the data from the device's address. I/O operations behave much like memory read and write operations, except that I/O usually involves more wait states.

Based on the CPU's ability to read and write data at a given port address, I/O ports can be grouped into five categories: read-only, write-only, read/write, dual I/O, and bidirectional.

A *read-only port* is an input port. If the CPU can only read the data from the port, then the data must come from some source external to the computer system. It's never a good idea to try to write to a read-only port because, although the hardware typically ignores such attempts, it can cause some devices to fail. A good example of a read-only port is the status port on the original IBM PC's parallel printer interface. Data from this port specifies the current status of the printer, while the hardware ignores any data written to this port.

A *write-only port* is always an output port. Data written to such a port is available for use by an external device. Attempting to read data from a write-only port generally returns whatever garbage value happens to be on the data bus, so your programs shouldn't depend on the meaning of such values. An output port typically uses a latch device to hold data to be sent to the outside world. When a CPU writes to a port address associated with an output latch, the latch stores the data and makes it available on an external set of signal lines (see Figure 12-1).

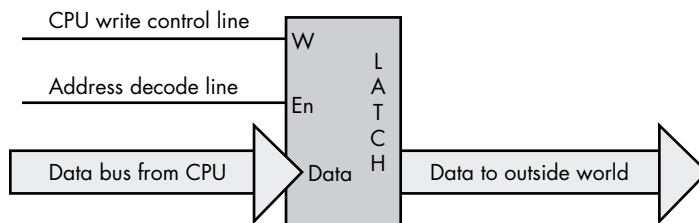


Figure 12-1: A typical write-only port

A perfect example of an output port is a parallel printer port. The CPU typically writes an ASCII character to a byte-wide output port that connects to the DB-25F connector on the back of the computer's case. A cable transmits this data to the printer, where it arrives on the printer's input port (from the printer's perspective, it is reading the data from the computer system). A processor inside the printer typically converts this ASCII character to a sequence of dots that it prints on paper.

Output ports can be write-only or read/write. The port in Figure 12-1, for example, is a write-only port. Because the outputs on the latch do not loop back to the CPU's data bus, the CPU can't read the data the latch contains. Both the address decode line (En) and the write control line (W) must be active for the latch to operate. If the CPU tries to read the data located at the latch's address, the address decode line is active but the write control line is not, so the latch does not respond to the read request.

A *read/write port* is an output (write-only) port as far as the outside world is concerned. However, as the name implies, the CPU can also read data from such a port—specifically, it reads the data that was last written to the port. Doing so does not affect the data presented to the external peripheral device.¹ Figure 12-2 illustrates a read/write port.

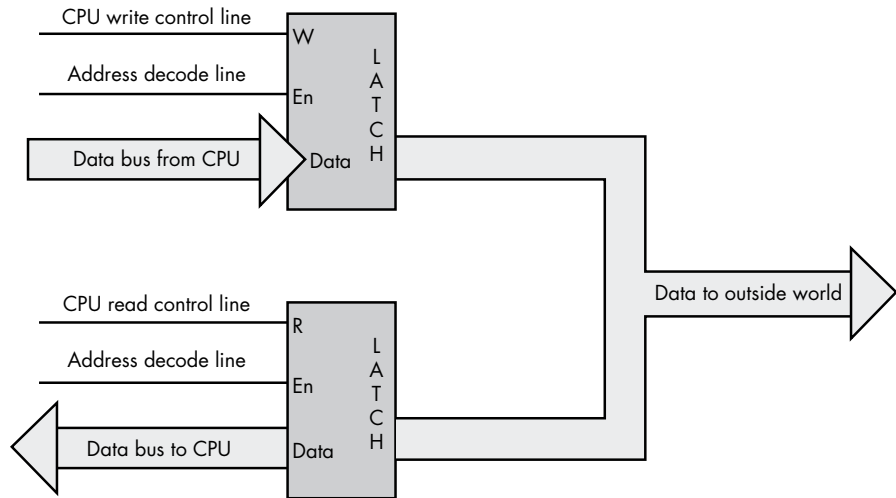


Figure 12-2: A read/write port

As you can see, the data written to the output port loops back to a second latch. Placing the address of these two latches on the address bus asserts the *address decode lines* on both latches. Therefore, to select between the two latches, the CPU must also assert either the read line or the write line. Asserting the read line (which happens during a read operation) will enable the lower latch. This places the data previously written to the output port on the CPU's data bus, allowing the CPU to read that data.

The port in Figure 12-2 is not an input port—true input ports read data from external pins. Although the CPU can read data from this latch, the organization of this circuit simply allows the CPU to read the data it previously wrote to the port, thus saving the program from having to maintain this value in a separate variable. The data on the external connector is output only, and you can't connect real-world input devices to these signal pins.

1. Historically, “peripheral” meant any device external to the computer system itself. This book will use the modern form of this term to refer to any device that is not part of the CPU or memory.

A *dual I/O port* is also a read/write port, but when you read a dual I/O port, you read data from an external input device rather than the last data written to the output side of the port's address. Writing data to a dual I/O port transmits data to some external output device, just as writing to a write-only port does. Figure 12-3 shows how you could interface a dual I/O port with the system.

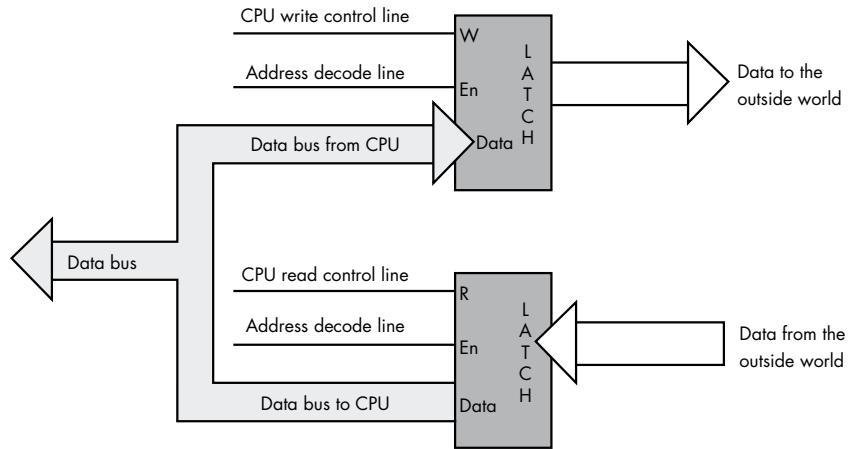


Figure 12-3: A dual I/O port

A dual I/O port is actually created with two ports—a read-only port and a write-only port—that share the same port address. Reading from the address accesses the read-only port, and writing to the address accesses the write-only port. Essentially, this port arrangement uses the read (R) and write (W) control lines to provide an extra address bit that specifies which of the two ports to use.

Finally, a *bidirectional port* allows the CPU to both read data from and write data to an external device. To function properly, a bidirectional port must pass various control lines, such as read and write enable, to the peripheral device so that the device can change the direction of data transfer based on the CPU's read/write request. In effect, a bidirectional port is an extension of the CPU's bus through a bidirectional latch or buffer.

Generally, a given peripheral device utilizes multiple I/O ports. The original IBM PC parallel printer interface, for example, uses three port addresses: a read/write I/O port, a read-only input port, and a write-only output port. The read/write data port allows the CPU to read the last ASCII character written through it. The input port returns control signals from the printer, which indicate whether the printer is ready to accept another character, offline, out of paper, and other statuses. The output port transmits control information to the printer. Later-model PCs substituted a bidirectional port for the data port, allowing data transfer from and to a device through the parallel port. The bidirectional data port improved performance for various devices such as disk and tape drives connected to the PC's parallel port. (Of course, modern PCs talk to printers over the USB port—that's quite a different animal from the hardware perspective, though.)

12.2 Other Ways to Connect Ports to the System

The examples thus far may have given you the impression that the CPU always reads and writes peripheral data using the data bus. However, while the CPU generally transfers the data it has *read* from input ports across the data bus, it doesn't always use the data bus to *write* data to output ports. In fact, a very common output method is to simply access a port's address directly without writing any data to it. Figure 12-4 illustrates a simple example of this technique using a set/reset (S/R) flip-flop.

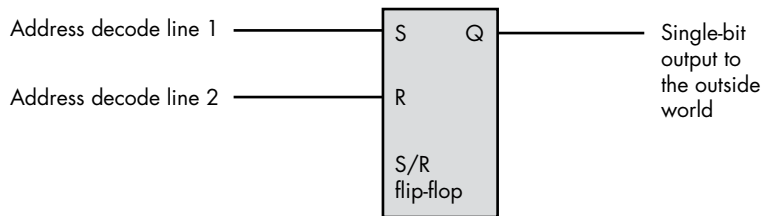


Figure 12-4: Outputting data to a port by directly accessing that port

In this circuit, an address decoder decodes two separate addresses. Any read or write access to the first address sets the output line to a 1; any read or write access to the second address sets the output line to a 0. This circuit ignores the data on the CPU's data lines, as well as the status of the read and write lines. The only thing that matters is that the CPU accesses one of these two addresses.

Another possible way to connect an output port to a system is to connect the read/write status lines to the data input of a D flip-flop. Figure 12-5 shows how you could design such a device.

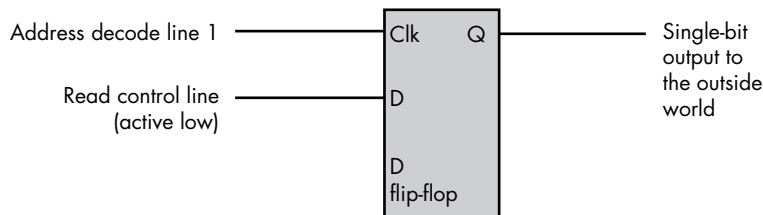


Figure 12-5: Outputting data using the read/write control as the data to output

In this diagram, any read of the port sets the output bit to 0, while any write to this port sets the output bit to 1 (the read control line will be HIGH when writing to the specified address).

These are only two examples of an amazing variety of designs that engineers have devised to avoid using the data bus (largely to reduce hardware costs or improve performance). However, unless otherwise noted, the remaining examples in this chapter presume that the CPU reads and writes data to and from an external device using the data bus.

12.3 I/O Mechanisms

There are three basic I/O mechanisms that computer systems use to communicate with peripheral devices: memory-mapped input/output, I/O-mapped input/output, and direct memory access (DMA). *Memory-mapped I/O* uses ordinary locations within the CPU's memory address space to communicate with peripheral devices. *I/O-mapped input/output* uses an address space separate from memory, as well as special machine instructions to transfer data between that I/O address space and the outside world. *Direct memory access (DMA)* is a special form of memory-mapped I/O where the peripheral device reads and writes data located in memory without CPU intervention. Each I/O mechanism has its own set of advantages and disadvantages, as we will discuss in this section.

Usually, the hardware system designer determines how a device connects to a computer system; programmers have little control over this decision. Nevertheless, by paying attention to the costs and benefits of the I/O mechanism used for communication between the CPU and the peripheral device, you can choose code sequences that will maximize I/O performance within your applications.

12.3.1 Memory-Mapped I/O

A memory-mapped peripheral device is connected to the CPU's address and data lines exactly like regular memory, so whenever the CPU writes to or reads from the address associated with the peripheral device, the CPU transfers data to or from the device. This mechanism has several benefits and only a few disadvantages.

The principle advantage of a memory-mapped I/O subsystem is that the CPU can use any instruction that accesses memory, such as `mov`, to transfer data between the CPU and a peripheral. For example, if you're trying to access a read/write or bidirectional port, you can use an 80x86 *read/modify/write* instruction, like `add`, to read the port, manipulate the value, and then write data back to the port, all with a single instruction. Of course, if the port is read-only or write-only, such an instruction will be of little use.

The big disadvantage of memory-mapped I/O devices is that they consume addresses in the CPU's memory map. Every byte of address space that a peripheral device consumes is one less byte available for installing actual memory. Generally, the minimum amount of space you can allocate to a peripheral (or block of related peripherals) is a page of memory (4,096 bytes on an 80x86). Fortunately, a typical PC has only a couple dozen such devices, so this usually isn't much of a problem. However, it can become a problem with some peripheral devices, like video cards, that consume a large chunk of the address space. Some video cards have between 1GB and 32GB of on-board memory that they map into the memory address space, which means that the 1GB to 32GB address range consumed by such a card is not available to the system for use as regular RAM (though this is hardly a concern on a 64-bit processor).

I/O AND THE CACHE

The CPU cannot cache values intended for memory-mapped I/O ports. Caching data from an input port would mean that subsequent reads of the port would access the value in the cache rather than the port data, which could be different. Similarly, with a write-back cache mechanism, some writes might never reach an output port because the CPU might save up several writes in the cache before sending the last one to the actual I/O port. In order to avoid these potential problems, we need some mechanism to tell the CPU not to cache accesses to certain memory locations.

The solution is found in the CPU's memory management subsystem. The 80x86's page table entries, for example, contain a flag that the CPU can use to determine whether it is okay to map data from a page in memory to the cache. If this flag is set one way, the cache operates normally; if the flag is set the other way, the CPU does not cache accesses to that page.

12.3.2 I/O-Mapped Input/Output

As noted previously, I/O-mapped input/output uses a special I/O address space separate from the normal memory space, coupled with special machine instructions to access device addresses. For example, the 80x86 CPUs provide the *in* and *out* instructions specifically for this purpose. These instructions behave like *mov* except that they transmit data to and from the special I/O address space rather than the normal memory address space. Typically, processors that provide I/O-mapped input/output capabilities use the same physical address bus to transfer both memory addresses and I/O device addresses. Additional control lines differentiate between addresses that belong to the normal memory space and those that belong to the special I/O address space. This means that such CPUs could use both I/O-mapped input/output or memory-mapped I/O. Therefore, if the number of I/O-mapped locations in the CPU's address space is insufficient, a hardware designer can always use memory-mapped I/O instead (as a video card does on a typical PC).

In modern 80x86 PC systems that utilize the PCI bus (or later variants), special peripheral chips on the system's motherboard remap the I/O address space into the main memory space, allowing programs to access I/O-mapped devices using either memory-mapped or I/O-mapped input/output.

12.3.3 Direct Memory Access

Memory-mapped I/O subsystems and I/O-mapped subsystems are both forms of *programmed I/O*, as they require the CPU to move data between the peripheral device and memory. To store into memory a sequence of 10 bytes taken from a programmed I/O input port, the CPU must read each value from the input port and store it into memory.

However, processing data 1 byte (or word or double word) at a time via the CPU may be too slow for very high-speed I/O devices. Such devices generally have an interface to the CPU's bus so they can read and write memory directly—that is, without the CPU as an intermediary. Direct memory access (DMA) allows I/O operations to proceed in parallel with other CPU operations, which increases the overall speed of the system—unless the CPU and the DMA device both try to use the address and data buses at the same time. Concurrent processing occurs only if the bus is free for use by the I/O device, which happens when the CPU has a cache and is accessing cached code and data. Nevertheless, even if the CPU must halt and wait for a DMA operation to complete before beginning a different operation, the DMA approach is still much faster, because many of the bus operations are instruction fetches or I/O port accesses that don't occur during DMA operations.

A typical DMA controller consists of a pair of counters and other circuitry that interfaces with memory and the peripheral device. One of the counters serves as an address register, supplying an address on the address bus for each transfer. The second counter specifies the number of data transfers. The application initializes the DMA controller's address counter with the address of the block where it should begin transferring data. Each time the peripheral device wants to transfer data to or from memory, it sends a signal to the DMA controller, which places the value of the address counter on the address bus. In coordination with the DMA controller, the peripheral device places data on the data bus to write to memory during an input operation, or it reads data from the data bus, taken from memory, during an output operation.² After a successful data transfer, the DMA controller increments its address register and decrements the transfer counter. This process repeats until the transfer counter decrements to zero.

12.4 I/O Speed Hierarchy

Different peripheral devices have different data transfer rates. Some devices, like keyboards, are extremely slow compared to CPU speeds. Other devices, like solid-state disk drives, can actually transfer data faster than the CPU can process it. The appropriate programming technique for data transfer depends strongly on the transfer speed of the peripheral device involved in the I/O operation. Therefore, before discussing how to write the most appropriate code, we should establish some terminology to describe the different transfer rates of peripheral devices.

Low-speed devices Devices that produce or consume data at a rate much slower than the CPU is capable of processing. For the purposes of discussion, we'll assume that low-speed devices operate at speeds that are three or more orders of magnitude slower than the CPU.

2. Don't forget that "input" and "output" are from the perspective of the computer system, not the device. Hence, the device writes data during an input operation and reads data during an output operation.

Medium-speed devices Devices that transfer data at approximately the same rate as, or up to three orders of magnitude slower than, the CPU (accessing the device using programmed I/O).

High-speed devices Devices that transfer data faster than the CPU is capable of handling using programmed I/O.

The speed of the peripheral device determines the type of I/O mechanism used for the I/O operation. Clearly, high-speed devices must use DMA because programmed I/O is too slow. Medium- and low-speed devices can use any of the three I/O mechanisms for data transfer (though low-speed devices rarely use DMA because of the cost of the extra hardware involved).

With typical bus architectures, CPUs are capable of one transfer per microsecond or better. Therefore, high-speed devices are those that transfer data more rapidly than once per microsecond. Medium-speed transfers are those that involve a data transfer every 1 to 100 microseconds. Low-speed devices usually transfer data less often than once every 100 microseconds. Of course, these definitions for low-, medium-, and high-speed devices are system dependent. Faster CPUs with faster buses allow faster medium-speed operations.

Note that one transfer per microsecond is not the same as a 1MB-per-second transfer rate. A peripheral device can actually transfer more than 1 byte per data transfer operation. For example, when using the `80x86 in(dx, eax);` instruction, the peripheral device can transfer 4 bytes in one transfer. Therefore, if the device is capable of one transfer per microsecond, it can transfer 4MB per second using this instruction.

12.5 System Buses and Data Transfer Rates

In Chapter 6, you saw that the CPU communicates with memory and I/O devices using the system bus. If you've ever looked inside a computer or read the specifications for a system, you've probably seen terms like *PCI*, *ISA*, *EISA*, or even *NuBus* used to refer to the computer's system bus. In this section, we'll discuss how these different computer system buses relate to the CPU bus, and how they affect the performance of a system.

A single computer system often employs multiple buses. Therefore, a software engineer can choose which peripheral devices to use based upon their bus connections. Maximizing performance for a particular bus may require different programming techniques than for other buses. Although it's not possible to choose the buses a particular computer system employs, a software engineer can select among the available buses to improve an application.

Computer system buses like PCI (Peripheral Component Interconnect) and ISA (Industry Standard Architecture) define physical connectors inside a computer system. Specifically, they describe the set of electronic signals (*connector pins* on the bus), physical dimensions (that is, connector layouts and distances from one another), and a data transfer protocol for connecting different electronic devices. These buses are often extensions of the CPU's *local bus* (the address, data, and control lines), because many of the signals on the system buses are identical to the CPU's signals.

However, peripheral buses themselves are not necessarily identical to the CPU's bus—they may have additional or fewer signals compared to those on the CPU. For example, the ISA bus supports only 24 address lines compared with the Intel and AMD's x86-64 40 to 52 address lines.

Different peripheral devices are designed to use different peripheral buses. Figure 12-6 shows the organization of the PCI and ISA buses in a typical computer system.³

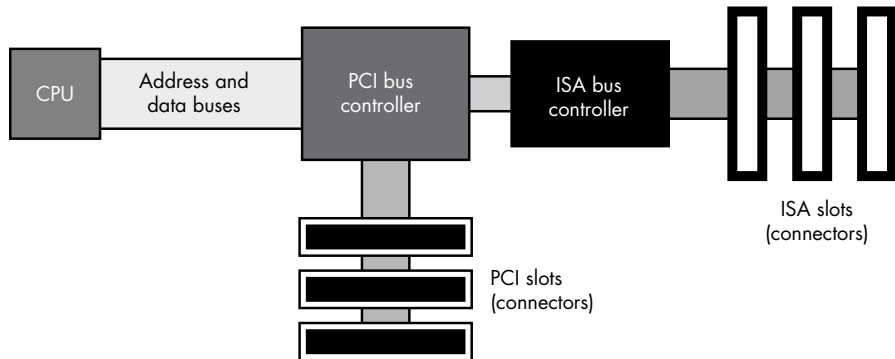


Figure 12-6: Connection of the PCI and ISA buses in a typical PC

Notice how the CPU's address and data buses connect to a PCI bus controller peripheral device, but not to the PCI bus itself. The PCI bus controller contains two sets of pins, providing a *bridge* between the CPU's local bus and the PCI bus. The signal lines on the local bus are not connected directly to the corresponding lines on the PCI bus; instead, the PCI bus controller acts as an intermediary, rerouting all data transfer requests between the CPU and the PCI bus.

Also note that the ISA bus controller is usually connected to the PCI bus controller, not directly to the CPU. This is typically for cost or performance reasons (there may be a limit to the number of devices that can connect directly to the CPU bus without additional buffering, for example).

The CPU's local bus usually runs at some fraction of the CPU's frequency. Typical local bus frequencies are currently 66 MHz, 100 MHz, 133 MHz, 400 MHz, 533 MHz, and 800 MHz, but they may become even faster. Usually, only memory and a few selected peripherals like the PCI bus controller sit on the CPU's bus and operate at this high frequency.

Because a typical CPU's bus is 64 bits wide and it's theoretically possible to achieve one data transfer per clock cycle, the CPU's bus has a maximum data transfer rate of 8 bytes times the clock frequency, or 800MB per second for a 100 MHz bus. In practice, CPUs rarely achieve the maximum data transfer rate, but they do achieve some percentage of it, so the faster the bus, the more data can move in and out of the CPU (and caches) in a given amount of time.

3. The ISA bus is the original IBM PC/AT bus. You won't see it very often on modern computer systems.

12.5.1 Performance of the PCI Bus

The PCI bus comes in several configurations. The base configuration has a 32-bit-wide data bus operating at 33 MHz. Like the CPU's local bus, the PCI bus is theoretically capable of transferring data on each clock cycle. This means that the bus has a theoretical maximum data transfer rate of 4 bytes times 33 MHz, or 132MB per second. In practice, though, the PCI bus doesn't come anywhere near this level of performance except in short bursts. Newer versions of the PCI-e offer up to 16 "lanes," allowing for much faster data transfer (largely for high-performance video cards).

Whenever the CPU wants to access a peripheral on the PCI bus, it must negotiate with other peripheral devices for the right to use the bus. This negotiation can take several clock cycles before the PCI controller grants the CPU access to the bus. If a CPU writes a double word per bus transfer, the negotiation time actually slows the transfer rate dramatically. The only way to achieve anywhere near the maximum theoretical bandwidth on the bus is to use a DMA controller and move blocks of data in *burst mode*. In burst mode, the DMA controller negotiates just once for the bus and then makes many transfers without giving up the bus between each one.

There are a couple of enhancements to the PCI bus that improve performance. Some PCI buses support a 64-bit-wide data path. This, obviously, doubles the maximum theoretical data transfer rate from 4 bytes per transfer to 8 bytes per transfer. Another enhancement is running the bus at 66 MHz, which also doubles the throughput. With a 64-bit-wide, 66 MHz bus, you would quadruple the data transfer rate of the baseline configuration. These optional enhancements to the PCI bus allow it to grow with the CPU as CPUs increase their performance. A high-performance version of the PCI bus, PCI-X, was available for a while, but it has largely been replaced by the PCI-e bus. PCI-e is a serial bus, transmitting data serially over a few data lines. However, it uses lanes to pass additional data in parallel. For example, a 16-lane PCI-e bus is 16 times faster than a single-lane variant.

12.5.2 Performance of the ISA Bus

The ISA bus is a carryover from the original PC/AT computer system. This bus is 16 bits wide and operates at 8 MHz. It requires four clock cycles for each bus cycle (a *bus cycle* is the time it takes to transfer one 16-bit word of data across the ISA bus). For this and other reasons, the ISA bus is capable of about only one data transmission per microsecond. With a 16-bit-wide bus, data transfer is limited to about 2MB per second. This is much slower than both the CPU's local bus and the PCI bus. Generally, the ISA bus is really only capable of supporting low-speed and medium-speed devices—like an RS-232 communications device, a modem, or a parallel printer interface—to the ISA bus. Most other devices, like disks, scanners, and network cards, are too fast for the ISA bus.

Accessing the ISA bus on most systems involves first negotiating for the PCI bus, but the PCI bus is so much faster than the ISA bus that this negotiation time has very little impact on the performance of peripherals on

the ISA bus. Therefore, connecting the ISA controller directly to the CPU's local bus wouldn't noticeably improve performance.

Fortunately, the ISA bus is thoroughly obsolete these days, and you won't find it on modern PCs. A few industrial PCs and SBCs (single-board computers) support ISA bus connections for legacy applications, but other than that the ISA bus is dead.

12.5.3 The AGP Bus

Video display (aka graphics) cards are very special peripherals that need maximum bus performance to ensure quick screen updates and fast graphic operations. Unfortunately, if the CPU has to constantly negotiate with other peripherals for the use of the PCI bus, graphics performance can suffer. To overcome this problem, video card designers created the *Accelerated Graphics Port (AGP)*, an interface between the CPU's local bus and the video display card that provides various control lines and bus protocols specifically designed for video display cards.

The AGP connection lets the CPU quickly move data to and from the video display RAM (see Figure 12-7).

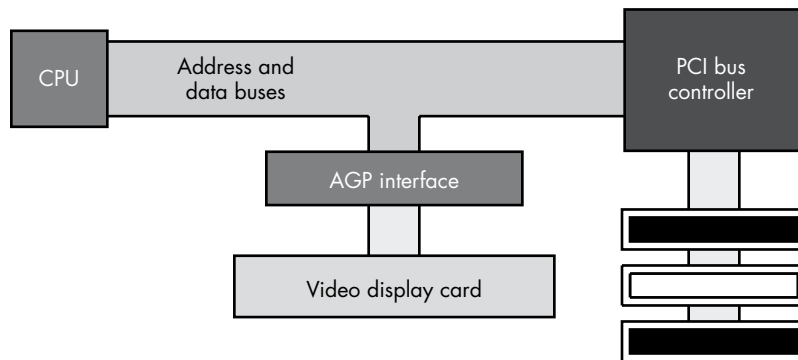


Figure 12-7: The AGP bus interface

Because there's only one AGP port per system, only one card can use the AGP slot at a time. The upside of this is that the system never has to negotiate for access to the AGP bus. However, by 2008 the performance of video cards surpassed that of the AGP bus. Most modern video cards use multilane PCI-e bus interfaces instead.

12.6 Buffering

If a particular I/O device produces or consumes data faster than the system is capable of transferring data to or from that device, the system designer has two choices: provide a faster connection between the CPU and the device, or slow down the rate of transfer between the two.

If the peripheral device is connected to a slow bus like ISA, a system designer can create a faster connection by switching to a wider bus like the 64-bit PCI, a faster bus (one with a higher frequency), or a higher-performance bus like PCI-e. System designers can also sometimes create a faster interface to the bus, as they did with the AGP connection.

The alternative—slowing down the transfer rate between the peripheral and the computer system—isn't always as bad an option as it might initially seem. Most high-speed devices don't transfer data to the system at a constant rate. Instead, they typically transfer a block of data rapidly and then sit idle for some time. Although the burst rate is higher than the CPU or memory can handle, the average data transfer rate is usually lower. If you can average out the high-bandwidth peaks and transfer some of the data when the peripheral is inactive, you can easily move data between the peripheral and the computer system without resorting to an expensive, high-bandwidth bus or connection.

The trick is to use memory on the peripheral side to buffer the data. The peripheral can rapidly fill this buffer with data during an input operation, and rapidly extract data from the buffer during an output operation. Once the peripheral device is inactive, the system either empties or refills the buffer at a sustainable rate. As long as the average data transfer rate of the peripheral device is below the maximum bandwidth the system supports, and the buffer is large enough to hold bursts of data going to and from the peripheral, this scheme lets the peripheral communicate with the system at a lower average data transfer rate.

Often, to save costs, the buffering takes place in memory on the CPU rather than on the peripheral device. In this case, it is often the software engineer's responsibility to initialize the buffer for a peripheral device. In some cases, neither the peripheral device nor the OS provides a buffer for the peripheral's data, so the application must do so in order to maintain maximum performance and avoid data loss. In other cases, the device or OS may provide a small buffer, but the application itself might not process the data often enough to avoid data overruns in the small buffer; in these situations, an application can create a larger buffer that is local to the application.

12.7 Handshaking

Many I/O devices cannot accept data at just any rate. For example, an i9-based PC is capable of sending several hundred million characters per second to a printer, but printers can't print that many characters each second. Likewise, an input device such as a keyboard will never transmit several million keystrokes per second to the system (because the keyboard operates at human speeds, not computer speeds). Because of these differences in capabilities, the CPU needs some way to coordinate data transfer between the computer system and its peripheral devices.

One common approach is to send and receive status bits on a port separate from the data port. For example, a printer could send a single bit to tell

the system whether it is ready to accept more data. Likewise, a single status bit in a different port could specify whether a keystroke is available at the keyboard data port. The CPU can test these bits prior to writing a character to the printer or reading a key from the keyboard.

Using status bits to indicate that a device is ready to accept or transmit data is known as *handshaking*, so named because the protocol is similar to two people signifying agreement with a handshake.

The following 80x86 assembly language program segment demonstrates how handshaking works:

```
mov( $379, dx );      // Initialize DX with the address of the status port.
repeat

    in( dx, al );      // Get the parallel port status into the AL register.
    and( $80, al );    // Clear z flag if the H0 bit is set.

until( @nz );         // Repeat until the H0 bit contains a 1.

// Okay to write another byte to the printer data port here.
```

This code fragment will continuously loop while the H0 bit of the printer status register (at input port \$379) contains 0 and will exit once the H0 bit is set (indicating that the printer is ready to accept data).

12.8 Timeouts on an I/O Port

One problem with the repeat..until loop in the previous section is that it could spin indefinitely as it waits for the printer to become ready to accept additional input. If someone turns the printer off or the printer cable becomes disconnected, the program could freeze up, forever waiting for the printer to become available. Usually, it's a better idea to inform the user when something goes wrong rather than allowing the system to hang. To do this, include a *timeout* period in the loop; once exceeded, the timeout causes the program to alert the user that something is wrong with the peripheral device.

You can expect some sort of response from most peripheral devices within a reasonable amount of time. For example, even in the worst case, most printers will be ready to accept additional character data within a few seconds of the last transmission. Therefore, something is probably wrong if 30 seconds or more has passed without the printer accepting a new character. A program written to detect this kind of problem typically pauses, asking the user to check the printer, and then resumes printing once the user indicates the problem is resolved.

Choosing a good timeout period is not an easy task. You must carefully balance the irritation of possible false alarms from the program with the pain of having it lock up for long periods when something actually is wrong. Both situations are equally annoying.

An easy way to create a timeout period is to count the number of times the program loops while waiting for a handshake signal from a peripheral. Consider the following modification to the repeat..until loop from the previous section:

```
mov( $379, dx );           // Initialize DX with the address of the status port.
mov( 30_000_000, ecx );    // Timeout period of approximately 30 seconds,
                           // assuming port access time is about 1 microsecond.
HandshakeLoop:

    in( dx, al );           // Get the parallel port status into the AL register.
    and( $80, al );         // Clear z flag if the H0 bit is set.

loopz HandshakeLoop;       // Decrement ECX and loop while ECX <> 0 and
                           // the H0 bit of AL contains a 0.

if( ecx <> 0 ) then

    // Okay to write another byte to the printer data port here.

else

    // We had a timeout condition if we get here.

endif;
```

This code will exit once the printer is ready to accept data or when approximately 30 seconds have expired. You might question the 30-second figure, since a software-based loop (counting down ECX to 0) should run at different speeds on different processors. However, the `in()` instruction reads a port on the bus, and that means this instruction will take approximately 1 microsecond to execute (I/O ports often inject lots of wait states). Hence, one million times through the loop will take about a second (plus or minus 50 percent, but close enough for our purposes). This is true almost regardless of the CPU frequency.

12.9 Interrupts and Polled I/O

Polling is the process of constantly testing a port to see if data is available. The handshaking loops of the previous sections provide good examples of polling—the CPU waits in a short loop, testing the printer port’s status value until the printer is ready to accept more data, and then the CPU can transfer more data to the printer. Polled I/O is inherently inefficient. If the printer in this example takes 10 seconds to accept another byte of data, the CPU spins, doing nothing productive for those 10 seconds.

In early personal computer systems, this is exactly how a program would behave. When a program wanted to read a key from the keyboard, it would poll the keyboard status port until a key was available. These early computers could not do other processing while waiting for the keyboard.

The solution to this problem is to use an *interrupt mechanism*. An interrupt is triggered by an external hardware event, such as the printer becoming ready to accept another character, that causes the CPU to interrupt its current instruction sequence and call a special *interrupt service routine (ISR)*. Typically, an ISR runs through the following sequence of events:

1. It preserves the current values of all machine registers and flags so that the interrupted computation can be continued later.
2. It does whatever operation is necessary to *service* the interrupt.
3. It restores the registers and flags to the values they had before the interrupt.
4. It resumes execution of the code that was interrupted.

In most computer systems, typical I/O devices generate an interrupt whenever they make data available to the CPU, or when they become able to accept data from the CPU. The ISR quickly processes the interrupt request in the background, allowing some other computation to proceed normally in the foreground.

Though ISRs are usually written by OS designers or peripheral device manufacturers, most OSes enable you to pass an interrupt to an application via *signals* or some similar mechanism. This allows you to include ISRs directly within an application. You could use this facility, for example, to have a peripheral device notify your application when its internal buffer is full and the application needs to copy data from the peripheral's buffer to an application buffer to prevent data loss.

12.10 Protected-Mode Operation and Device Drivers

If you're working on an ancient Windows 95 or 98 system, you can write assembly code to access I/O ports directly. The handshaking code shown earlier is a good example of this. However, modern versions of Windows and all versions of Linux and macOS employ a *protected mode* of operation. In this mode, direct access to devices is restricted to the OS and certain privileged programs. Standard applications, even those written in assembly language, are not so privileged. If you write a simple program that attempts to send data to an I/O port, the system will generate an illegal access exception and halt your program.

Linux won't allow just any program to access I/O ports; only programs with "superuser" (root) privileges can do so. For limited I/O access, it's possible to use the Linux `ioperm` system call to make certain I/O ports accessible from user applications. (For more details, read the man page on `ioperm`.)

If Linux, macOS, and Windows don't allow direct access to peripheral devices, how does a program communicate with these devices? Clearly, this *can* be done, because applications interact with real-world devices all the time. The answer is that these OSes permit specially written modules, known as *device drivers*, to access I/O ports. A complete discussion of writing device drivers is well beyond the scope of this book, but understanding how

they work may help you understand the possibilities and limitations of I/O under a protected-mode OS.

12.10.1 The Device Driver Model

A device driver is a special type of program that links with the OS. It must follow some specific protocols, and it must make some special calls to the OS that are not available to standard applications. Furthermore, in order to install a device driver in your system, you must have administrator privileges, because device drivers pose all kinds of security and resource allocation risks, and you can't leave your system vulnerable. Therefore, installation is not a trivial process, and application programs cannot load and unload drivers at will.

Fortunately, there are only a limited number of devices found on a typical PC, so you only need a limited number of device drivers. You would typically install a device driver in the OS at the same time you install the device, or, if the device is built into the PC, at the same time you install the OS. About the only time you'd really need to write your own device driver is when building your own device, or in unique cases where you need to take advantage of some device's capabilities that standard device drivers don't handle.

The device driver model works well with low-speed devices, where the OS and device driver can respond to the device much more quickly than it requires. The model is also great for use with medium- and high-speed devices where the system transmits large blocks of data to and from the device. However, the device driver model does have a few drawbacks, one being that it does not support medium- and high-speed data transfers that require substantial interaction between the device and the application.

The problem is that calling the OS is an expensive process. Whenever an application makes a call to the OS to transmit data to the device, it can potentially take hundreds of microseconds, if not milliseconds, before the device driver actually sees the application's data. If the interaction between the device and the application requires a constant flurry of bytes moving back and forth, there will be a big delay if each transfer has to go through the OS. For applications of this sort, you'll need to write a special device driver that can handle the transactions itself rather than continually returning to the application.

Because applications can't access devices directly (in modern OSes), all communication between them must take place through a device driver intermediary. The question, then, is how do applications communicate with device drivers?

12.10.2 Communication with Device Drivers

For the most part, communicating with a peripheral device under a modern OS is exactly like writing data to a file or reading data from a file. In most OSes, you open a "file" using a special filename like *COM1* (the serial port) or *LPT1* (the parallel port) and the OS automatically creates a connection to the specified device. When you are finished using the device, you

“close” the associated file, which tells the OS that the application is done with the device so other applications can use it.

Of course, most devices don’t support the same semantics as disk files. Some devices, like printers or modems, can accept a long stream of unformatted data, but others may require that you preformat the data into blocks and write the blocks to the device with a single write operation. The exact semantics depend upon the particular device. Nevertheless, the typical way to send data to a peripheral is to use an OS “write” function to which you pass a buffer containing some data, and the way to read data from a device is to call an OS “read” function to which you pass the address of some buffer into which the OS will place the data it reads.

But not all devices conform to these *stream-I/O* data semantics of file I/O, either. Therefore, most OSes provide a *device-control API* that lets you pass information directly to the peripheral’s device driver to handle the cases where a stream-I/O model fails.

Because it varies by OS, the exact details concerning the OS API interface are a bit beyond the scope of this book. Though most OSes use a similar scheme, they differ enough to make it impossible to describe them in a general way. So, for further details, consult the programmer’s reference for your particular OS.

12.11 For More Information

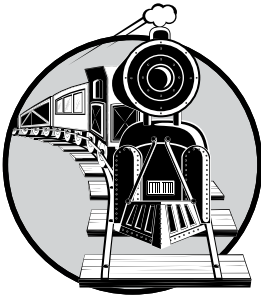
Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. “Chapter 13: I/O Systems.” In *Operating System Concepts*. 8th ed. Hoboken, NJ: John Wiley & Sons, 2009.

NOTE

Early editions of Patterson and Hennessy’s Computer Architecture: A Quantitative Approach provided a good chapter on I/O devices and buses; sadly, as it covered very old peripheral devices, the authors dropped the chapter rather than updating it in subsequent revisions. Internet searches seem to be the last place you can find consistent information on this subject (outside of this book, of course).

13

COMPUTER PERIPHERAL BUSES



System buses are not the only buses you'll find in a computer system. There are many specialized peripheral buses as well. This chapter discusses the SCSI, IDE/ATA, SATA, SAS, FibreChannel, Firewire, and USB buses that connect computers with various peripheral devices.

13.1 The Small Computer System Interface

The *Small Computer System Interface* (SCSI, pronounced “scuzzy”) is a peripheral interconnection bus used to connect high-speed peripheral devices to personal computer systems. Designed in the early 1980s, the SCSI bus was popularized by its introduction on the Apple Macintosh computer system in the mid 1980s. The original SCSI bus supported an 8-bit bidirectional data bus and was capable of transferring 5MB of data per second, which was considered high performance for hard-disk subsystems of that era. Although its early performance is quite slow by modern standards, SCSI has gone through

several revisions over the years and remains a high-performance peripheral interconnection system. At the height of its popularity, these older SCSI devices were capable of transferring 320MBps (megabytes per second).

Although the SCSI interconnection system is most commonly used for disk drive subsystems, SCSI was designed to support a whole host of PC peripherals using a cable connection. Indeed, as SCSI became popular during the late 1980s and into the 1990s, you could find printers, scanners, imaging machines, phototypesetters, network and display adapters, and many other devices interfacing with the SCSI bus.

However, the prevalence of SCSI as a general-purpose peripheral bus has diminished since the emergence of the USB, FireWire, and Thunderbolt peripheral connection systems. Except for very high-performance disk drive subsystems and some very specialized peripheral devices, few new peripherals use the interface. To understand why SCSI's popularity waned, let's look at the problems SCSI users have faced over the years.

13.1.1 Limitations

When SCSI was first introduced, the SCSI bus supported concurrent connection of the SCSI adapter card and up to seven actual peripheral devices. To connect multiple devices, you ran a cable from the host controller card to the first peripheral device. To connect a second device, you ran a cable from a second connector on the first device to the second device. To connect a third device, you ran a cable from a separate connector on the second device to the third device, and so on. At the end of this “daisy chain” of devices, you attached a special terminating device to the last connector of the last peripheral device. Without the special “terminator” at the end of the SCSI chain, many SCSI systems would work unreliably, if at all.

As a “convenience” to their customers, many peripheral manufacturers built the terminating circuitry into their devices. Unfortunately, connecting multiple terminators in the middle of the SCSI chain was just as bad as not having a terminator at all. Though most manufacturers who designed the terminating circuitry into their peripherals often provided an option to disable the terminator, some did not. Ensuring that those devices with the active terminator circuitry were at the end of the SCSI chain was often cumbersome, and even if a device provided an option to enable or disable the terminator, knowing the appropriate DIP switch settings was challenging if the documentation wasn't handy. As a result, many computer owners had problems with a chain of SCSI devices not working properly in their system.

On the original SCSI bus, the computer system owner had to assign each device one of eight numeric “addresses” from 0 to 7, with address 7 generally reserved for the host controller card. If two devices in the SCSI chain had the same address, they wouldn't operate properly. This made moving SCSI peripherals from one computer system to another somewhat difficult, because the address of the device being moved was usually already taken by another device on the new system.

The original SCSI bus had other limitations as well. First, it supported only seven peripheral devices. When SCSI was first designed, this wasn't usually a problem because common SCSI peripherals like hard drives and scanners were very expensive, costing thousands of dollars each. Connecting more than seven devices wasn't something your average computer owner would have done back then. As the price of hard drives and other SCSI peripherals came down, however, the seven-peripheral limit became burdensome.

Second, SCSI was not *hot-swappable*; that is, you couldn't unplug or connect a peripheral device while the power was on. Doing so could cause electrical damage to the SCSI controller, the peripheral, or even some other peripheral on the SCSI bus. As SCSI peripherals became more affordable and people began connecting multiple devices to their computer systems, the desire to unplug a device from one system and plug it into another grew, but SCSI did not support that capability.

13.1.2 Improvements

Despite these drawbacks, SCSI's popularity grew. To maintain that popularity, SCSI was modified over time to improve its functionality. SCSI-2, the first modification, increased the speed from 5 MHz to 10 MHz, thus doubling the data transfer rate on the bus. This was necessary because the speed of high-performance devices like disk drives improved so much that the original SCSI was actually slowing them down. Next, expanding the size of the bidirectional SCSI data bus from 8 bits to 16 bits not only doubled the data transfer rate from 10MBps to 20MBps, but also increased the number of peripherals you could place on the bus from 7 to 15. Variations of SCSI-2 were known as *Fast SCSI* (10 MHz), *Wide SCSI* (16 bits), and *Fast and Wide SCSI* (16 bits at 10 MHz).

It should come as no surprise that SCSI-3 followed SCSI-2. SCSI-3 offers a veritable smorgasbord of different connection options while maintaining compatibility with the older standards. Although SCSI-3 (using names like Ultra, Ultra-Wide, Ultra2, Wide Ultra2, Ultra3, and Ultra320) still operates as a 16-bit bus in the parallel cable mode, and still supports a maximum of 15 peripherals, it vastly increased the operating speed of the bus and the maximum permissible physical distance across which SCSI peripherals could be chained. In short, SCSI-3 operates at speeds of up to 160 MHz, allowing the SCSI bus to transfer data in bursts up to 320MBps (that is, faster than many PCI bus interconnects!).

SCSI was originally a parallel interface. Today, it supports four different interconnection standards: SCSI Parallel Interface (SPI), Serial SCSI across FireWire, Fibre Channel Arbitrated Loop, and Serial-Attached SCSI (SAS). The SPI is the original standard that most people associate with SCSI. SCSI parallel cables contain either 8 or 16 data lines, depending on the type of SCSI interface in use. This makes SCSI cables bulky, heavy, and expensive. The parallel SCSI interface also limits the maximum length of the SCSI chain in the system to just a few meters. These concerns, especially

the economic ones, are why modern computer systems use SCSI peripherals only when they require extremely high performance.

Note that the computer system doesn't own the SCSI bus and doesn't necessarily direct the traffic between various peripherals on the bus. SCSI is a true *peer-to-peer* bus, and any two peripherals on it may communicate with each other. Indeed, it's possible (though unusual) for two computer systems to share the same SCSI bus.

This peer-to-peer operation can improve the performance of the overall system tremendously. To illustrate this point, consider a tape backup system. In practice, most tape backup programs read a block of data from a disk drive into the computer's memory and then write that block of data from the computer's memory to the tape drive. On the SCSI bus (in theory, at least), it's possible to have the tape and disk drives communicate directly with each other. The tape backup software would send two commands, one to the disk drive and one to the tape drive, telling the disk drive to transfer the block of data directly to the tape drive rather than going through the computer system. Not only does this reduce the number of transfers across the SCSI bus by half, speeding up the transfer, but it also frees up the computer's CPU to do other things. In reality, few tape backup systems work this way, but there are many examples where two peripherals communicate across the SCSI bus without using the computer as an intermediary. Software that programs SCSI peripherals to operate this way (rather than running the data through the computer's memory) is a prime example of great programming.

13.1.3 SCSI Protocol

SCSI is not only an electrical interconnection, but a *protocol* as well. You don't communicate with a SCSI peripheral device by writing some data to a couple of registers on the SCSI interface card, sending that data down the SCSI cable to the peripheral device. Instead, you build up a data structure in memory containing a SCSI command, command parameters, any data you want to send to the SCSI peripheral, and possibly a pointer with the memory address where the SCSI controller should store any data the peripheral device returns. Once you construct this data structure, you normally provide the SCSI controller with the data structure's address, and the SCSI controller then fetches the command from system memory and sends it to the appropriate peripheral device on the SCSI bus.

13.1.3.1 SCSI Command Set

As SCSI hardware has evolved over the years, so has the SCSI protocol—or the SCSI *command set*. SCSI was never intended to serve as just a hard-disk interface, and the breadth of peripherals that it supports has steadily increased over time with the advent of new types of computer peripherals. To accommodate these new and unanticipated uses for the SCSI bus, SCSI's designers created a device-independent command protocol that could be easily extended as new devices were invented. Contrast this with certain

device interfaces, such as the original Integrated Disk Electronics (IDE) interface, which was suitable only for disk drives.

The SCSI protocol transmits a packet containing the peripheral's address, the command, and the command's data. The SCSI-3 standard has roughly grouped these commands into the following classes:

SCSI Controller Commands (SCC) Controller commands for RAID arrays

SCSI Enclosure Services (SES) Commands Enclosure services commands

SCSI Graphics Commands (SGC) Graphics commands for printers

SCSI Block Commands (SBC) Hard-disk interface commands

Management Server Commands (MSC) Commands for converting between SCSI protocols

Multimedia Commands (MMC) Multimedia commands for devices such as DVD drives

Object-based Storage Device (OSD) Commands Commands for managing how objects are allocated, placed, and accessed

SCSI Primary Commands (SPC) Primary commands

Reduced Block Commands (RBC) Commands for simplified hard-drive subsystems

SCSI Stream Commands (SSC) Stream commands for tape drives

Although the SCSI commands themselves are standardized, the actual interface to the SCSI host controller is not. Different host controller manufacturers use different hardware to connect their SCSI controller chips to the host computer system, so how you talk to a SCSI controller chip depends on the particular host controller device. Because SCSI controllers are *very* complex and difficult to program, and because there is no “standard” SCSI interface chip, programmers are faced with having to write several different variants of their software to control SCSI devices.

13.1.3.2 SCSI Device Drivers

To correct this situation, SCSI host controller manufacturers like Adaptec have created specialized device driver modules that provide a uniform interface to their devices. Rather than writing data directly to a SCSI chip, a programmer creates an in-memory data structure with SCSI commands to be placed on the SCSI bus, calls the device driver software, and lets the device driver transfer the SCSI commands to the SCSI bus. There are several benefits of this approach:

- It frees the programmer from having to learn the complexities of each particular host controller.
- It allows different manufacturers to provide a compatible interface to their SCSI controller devices.

- It allows manufacturers to create a single optimized driver that properly supports the capabilities of their device, rather than prompting individual programmers to write (possibly mediocre) code for the device.
- It allows manufacturers to change the hardware of future versions of their device without destroying compatibility with existing software.

This concept was carried forward into modern OSes. Today, SCSI host controller manufacturers write *SCSI miniport drivers* for OSes like Windows. These miniport drivers provide a hardware-independent interface to the host controller so that the OS can simply say, “Here is a SCSI command. Put it on the SCSI bus.”

13.1.4 SCSI Advantages

One big advantage of the SCSI interface is that it provides parallel processing of SCSI commands. That is, a host system can place several different SCSI commands on the bus, and different peripheral devices can process those commands simultaneously. Some devices, like disk drives, can even accept multiple commands at once and process them in the order that is most efficient. As an example, suppose that a disk drive is currently near block 1,000. If the system sends block read requests for blocks 5,000; 4,560; 3,000; and 8,000; the disk controller can rearrange these requests and satisfy them in the most efficient order (probably 3,000; 4,560; 5,000; and then 8,000) as it moves the read/write head across the surface of the disk. This results in a big performance improvement on multitasking OSes that process requests for disk I/O from several different applications simultaneously.

SCSI is also a great interface for RAID systems because SCSI is one of the few disk controller interfaces that supports a large number of drives on the same interface.

The original SPI (parallel SCSI) is all but dead. Even SCSI over FireWire is almost gone (as is FireWire). However, today SCSI still lives on in the form of SAS (Serial-Attached SCSI). Very-high-performance hard-disk drives use the SAS command set (rather than the standard SATA command set). The highest-performing RAID systems are still built around SAS drives.

The SCSI command set is very powerful, and it is designed for high-performance applications. It is sufficiently large and complex that space limitations prevent its inclusion here. Readers interested in a deeper look at SCSI programming should refer to *The Book of SCSI*, 2nd ed., by Gary Field, Peter M. Ridge et al. (No Starch Press, 2000). The complete SCSI specifications appear at various sites on the web. A quick search for “SCSI specifications” should turn up several copies.

13.2 The IDE/ATA Interface

Although SCSI is very high performance, it is also expensive. A SCSI device requires a sophisticated and fast processor in order to handle all the operations that are possible on the SCSI bus. Furthermore, because SCSI devices can operate on a peer-to-peer basis (that is, one peripheral may talk to

another without intervention from a host computer system), each SCSI device must carry around a considerable amount of sophisticated software in ROM on the device's controller board. Adding all the extra functionality needed to support full SCSI when all you want to do is to attach a single hard disk to a personal computer system is overkill. The *Integrated Drive Electronics (IDE)* interface was an effort to provide a bare-bones, low-cost mass storage option.

The idea behind the IDE interface was to lower the cost of the disk drive by using the host computer's CPU to do the processing (SCSI used embedded CPUs to handle a lot of the work). Because the PC's CPU was usually idle (during SCSI transfers) anyway, this seemed like a good use of resources. IDE drives, because they were often hundreds of dollars less than SCSI drives, became incredibly popular on PC systems. The much lower cost of the IDE interface and of IDE drives ensured its popularity.

Because the original IDE specification was geared specifically to hard-disk drives and was not particularly well suited for other types of storage devices, the committee that designed the IDE interface went back to work and developed the *Advanced Technology Attachment with Packet Interface (ATAPI)*, which is usually shortened to *ATA*. Like SCSI, the ATA standard has gone through several revisions and improvements over the years. The ATAPI specification (in its eighth version as of 2013) extends IDE to support a wide range of mass storage devices, including tape drives, zip drives, CD-ROMs, DVDs, removable cartridge drives, and more. In order to extend the IDE interface to support all these different storage devices, ATAPI's designers adopted a packet command format that is very similar to—in some cases, identical to—the SCSI packet command format.

In modern protected-mode OSes like Windows or Linux, however, an application programmer is never allowed to talk directly to the hardware. In theory, it would be possible to write a miniport driver for IDE to simulate how SCSI works. In practice, though, the OS vendor generally supplies a software library that provides an *application programming interface (API)* to the IDE/ATAPI devices. The application programmer can then make function calls to the API, passing appropriate parameters, and the underlying library routines take care of the remaining tasks associated with actually talking to the hardware.

Programming ATAPI devices in a modern system is quite similar to programming SCSI devices. You load up a memory-based data structure with a command code and a set of parameters, and then pass the memory structure to a driver library function that passes the data across ATAPI to the target storage device. If such a low-level library is not available, and your OS allows it, you can program the ATAPI device to grab this data (generally using DMA on modern systems).

The full ATAPI specification is almost 500 pages long, so we don't have sufficient space to cover it here. If you're interested in a more detailed look at IDE/ATAPI, search for "ATAPI specifications" online.

Modern machines use a serial ATA (SATA) controller. This is a high-performance serial version of the venerable IDE/ATAPI parallel interface. However, to the programmer, it looks exactly like ATAPI.

13.2.1 The SATA Interface

As time passed, hard drives became sufficiently fast that the IDE/ATA interface was reducing drive performance. *Serial AT Attachment* (SATA) and, later, SATA-II and SATA-III, provided several advantages over the parallel IDE/ATA (often shortened to *PATA*, for “Parallel ATA”). Whereas PATA was capable of running at 133MBps, SATA-I, II, and III were capable of transferring data at 1.5Gbps (gigabits per second; 150MBps), 3.0Gbps (300MBps), and 6.0Gbps (600MBps), respectively, though few (RAID) systems even come close to achieving these data transfer rates. SATA also offered other advantages over PATA, including smaller cables (7 conductors rather than 40 or 80) and hot swapping. Today, most hard-disk drives connecting to PCs use the SATA interface (and most of the others use SAS, which is effectively SCSI over SATA, or Fibre Channel interfaces).

13.2.2 Fibre Channel

Fibre Channel is a very high-performance transport mechanism (up to 128Gbps). While it is a generic network protocol for large mainframe computers, one of its predominant uses is to connect very high-performance disk arrays to computer system (usually servers). For disk drive use, Fibre Channel transports SCSI commands across the Fibre Channel cabling. So, the 1980s SCSI interface lives on today in Fibre Channel, still the highest-performance disk interface protocol.

13.3 The Universal Serial Bus

The *Universal Serial Bus* (USB) is a mechanism that allows you to use a single interface to connect a wide variety of peripheral devices to a PC, similar to SCSI. The USB supports *hot-pluggable devices*, meaning you can plug and unplug devices without shutting down the power or rebooting your machine, and it supports *plug-and-play devices*, meaning the OS will automatically load a device driver, if available, once you plug in a device. This flexibility comes at a cost, however. Programming devices on the USB is considerably more complex than programming a serial or parallel port. You cannot communicate with USB peripherals by reading or writing a few device registers.

13.3.1 USB Design

To understand the motivation behind USB, consider the situation PC users faced when Windows 95 first arrived, nearly 14 years after the introduction of the IBM PC. IBM designed its PC with a variety of peripheral interconnects that were common on PCs and minicomputers in the late 1970s. However, the IBM designers didn’t anticipate (or allow for) the wide variety of peripheral devices that people would invent to attach to PCs in the following decades. They also did not count on any individual PC owners connecting more than a few different peripheral devices to their machines.

Certainly, three parallel ports, four serial ports, and a single hard-disk drive should have been sufficient!

By the time Windows 95 was introduced, people were connecting their PCs to all kinds of devices, including sound cards, video digitizers, digital cameras, advanced gaming devices, scanners, telephones, mice, digitizing tablets, SCSI devices, and literally hundreds of other devices the original PC's designers hadn't dreamed of. The creators of these devices interfaced their hardware to the PC using peripheral I/O port addresses, interrupts, and DMA channels that were originally intended for other devices. The problem with this approach was that there were a limited number of port addresses, interrupts, and DMA channels, and a large number of devices competing for them. As a workaround, the device manufacturers added "jumpers" to their cards that would allow the purchaser to select from a small set of different port addresses, interrupts, and DMA channels, to alleviate conflicts with other devices.

Creating a conflict-free system was a complex process, though, and it was impossible to achieve with some combinations of peripherals. In fact, one of the big selling points of the Apple Macintosh during this period was that you could easily connect multiple peripheral devices without worrying about device conflicts. What was needed was a new peripheral connection system that supported a large number of devices without conflicts. USB was the answer.

USB allows the connection of up to 127 devices simultaneously by using a 7-bit address. USB reserves the 128th slot, address 0, for autoconfiguration purposes. In real life, it's doubtful that you'd ever successfully connect so many devices to a single PC, but it's good to know that USB has a fair amount of potential for growth, unlike the original PC.

Despite the name, USB isn't a true "bus" in the sense of allowing several devices to communicate with one another. Instead, the USB is a controller/peripheral connection, with the PC always acting as controller. This means, for example, that a digital camera can't talk directly to a printer across the USB. To transmit information from the camera to the printer, both of which are connected to a PC, the camera must first send its data to the PC before the PC can pass the data along to the printer. The PCIe, ISA, FireWire (IEEE 1394), and Thunderbolt buses allow two devices to communicate peer-to-peer (that is, independent of the host's CPU), but USB wasn't designed to support this method of communication (to keep down the cost of peripherals and the USB interface chips they contain).¹

USB also keeps peripheral costs down by moving as much complexity as possible to the host (PC) side of the connection. The thinking here is that the PC's CPU will offer much higher performance than the low-cost microcontrollers found in most USB peripheral devices. This means that writing software to be embedded in a USB peripheral isn't much more work than

1. Recently, the USB Interface Group (or USB-IF) has defined an extension to the USB, known as *USB On-the-Go*, that allows a limited amount of (pseudo-)peer-to-peer operation. Rather than supporting true peer-to-peer operation, this scheme allows different peripherals to take turns being the master on the USB.

using another interface. On the other hand, writing USB software on the host side is very complex—so complex, in fact, that it isn’t realistic to expect programmers to do so.

Instead, the OS supplier must provide a USB host controller *stack* that enables communication with USB devices, and most application programmers talk to those devices using the OS’s device driver interface. Even programmers who need to write custom USB device drivers for their particular device don’t talk directly to the USB hardware. Instead, they make OS calls to the USB host controller stack with requests for their particular device. Because a typical USB host controller stack is generally around 20,000 to 50,000 lines of C code and requires several years of development, there’s little chance of programming USB devices on a system that doesn’t provide a native USB stack (such as MS-DOS).

13.3.2 USB Performance

The initial USB design supported two different types of peripherals—slow and fast—to support devices with different price points. Slow devices could transfer up to 1.5Mbps (megabits per second) across the USB, while fast devices were capable of transferring up to 12Mbps (1.5MBps). Cost-sensitive devices could be built inexpensively as low-speed devices. Non-cost-sensitive devices could use the 12Mbps data rate.

The USB 2.0 specifications added a high-speed mode supporting up to 480Mbps data transfer rates (60MBps), at considerable extra complexity and cost. USB 3.0 upped the performance to 635MBps (super-speed). Finally, the USB 3.1 and USB-C (Thunderbolt 3) interfaces bumped the speed up to 5GBps (gigabytes per second; SuperSpeed), 10GBps (SuperSpeed+), and 40GBps, respectively. USB 4.0 is expected to be capable of up to 80GBps.

USB does not dedicate the entire available bandwidth to one peripheral. Instead, the host controller stack *multiplexes* the data on the USB, effectively giving each peripheral a “time slice” of the bus. The USB operates with a 1-millisecond clock. At the start of each millisecond period, the USB host controller begins a new USB *frame*, and during a frame, each peripheral may transmit or receive a packet of data. Packets vary in size, depending on the speed of the device and the transmission time, but typically contain between 4 and 64 bytes of data. If you’re transferring data between four peripherals at an equal rate, you’d typically expect the USB stack to transmit one packet of data between the host and each peripheral in a *round-robin* fashion, taking care of the first peripheral first, the second peripheral second, and so on. Like time slicing in a multitasking OS, this data transfer mechanism gives the appearance of transferring data concurrently between the host and every USB peripheral, even though there can be only one transmission on the USB at a time.

Although USB provides a very flexible and expandable system, because the bandwidth on the bus is shared between all attached peripherals, it can slow devices down. For example, if you connect two disk drives to the USB and access both drives simultaneously, the two drives must share

the available bandwidth on the USB. For USB 1.x devices, this produces a noticeable speed degradation. For USB 2.x devices, the available bandwidth is sufficiently high (typically higher than what two disk drives can sustain) that you won't notice the performance degradation. For USB 3.x (and later) and USB-C, the performance is as high as many native bus controllers. (For example, Thunderbolt-3/USB-C provides a transport mechanism for the PCI bus and SCSI.) Theoretically, you could use multiple host controllers to provide multiple USB buses in a system (with full bandwidth available on each bus), but this addresses only part of the performance problem.

Another performance consideration is the overhead of the USB host controller stack. Although the USB 1.x hardware may be capable of 12Mbps bandwidth, there is some dead time—that is, time during which no transmission takes place on the USB—because the host controller stack takes a while to set up data transfers. In some USB systems, you can achieve at most *half* the theoretical USB bandwidth, because the host controller stack uses so much of the available CPU time setting up the transfer and moving data around. On some embedded systems using slower processors (such as 486, StrongArm, or MIPS) running an embedded USB 1.x host controller device, this can be a real problem.

If a particular host controller stack is incapable of maintaining the full USB bandwidth, it usually means that the CPU can't process USB information as fast as the USB produces it, because the CPU's processing capabilities are saturated—and no time is available for other computations, either. Remember, USB leaves all the complex computations for the host controller on the USB, and executing code in the USB stack on the host requires CPU cycles. It's quite possible for the host controller to get so involved processing USB traffic that overall system performance for non-USB traffic suffers.

Fortunately, on PCs with USB 2.x controllers, the host controller consumes only a small percentage of the USB bandwidth. When USB-3 and USB-C came along, USB hardware began supporting other transmission protocols, such as SCSI and PCI, eliminating many of the performance issues associated with USB.

13.3.3 Types of USB Transmissions

The USB protocol supports four different types of data transmissions: control, bulk, interrupt, and isochronous. The peripheral manufacturer, not the application programmer, determines the data transfer mechanism between the host and a given peripheral device. That is, if a device uses the isochronous data transfer mode to communicate with the host PC, a programmer can't decide to use bulk transfers instead. The application program may not even be aware of the underlying transmission scheme, as long as the software can handle the rate at which the device produces or consumes the data.

USB generally uses *control* transmissions to initialize a peripheral device by reading and writing data from and to a peripheral's registers. For example, if you have a USB-to-serial converter device, you would typically use control transfers to set the baud rate, number of data bits, parity, number of stop bits, and so on, just as you would store data into the 8250 SCC's

register set.² USB guarantees correct delivery of control transmissions and also guarantees that at least 10 percent of the USB bandwidth is available for control transmissions to prevent *starvation*, a situation where a particular transmission never occurs because some higher-priority transmission is always taking place.

USB *bulk* transmissions are used to transmit large blocks of data between the host and a peripheral device. Bulk transmissions are available only on full-speed (12Mbps), high-speed (480Mbps), and super-speed (USB 3/USB-C) devices, not on low-speed ones. On full-speed devices, a bulk transmission generally carries between 4 and 64 bytes of data per packet; on high- and super-speed devices, you can transmit up to 1,023 bytes per packet. USB guarantees correct delivery of a bulk packet between the host and the peripheral device, but it does not guarantee timely delivery. If the USB is handling a large number of other transmissions, it may take a while for a bulk transmission to complete. In theory, a bulk transmission might never occur if the USB is sufficiently busy with the right combination of isochronous, interrupt, and control transmissions. In practice, however, most USB stacks do set aside a small amount of guaranteed bandwidth for bulk transmissions (generally about 2 to 2.5 percent) to prevent starvation.

USB intends bulk transmissions to be used by devices that need to transmit a fair amount of data correctly, but not necessarily quickly. For example, when you're transferring data to a printer or between a computer and a disk drive, correct transfer is far more important than timely transfer. Sure, it may be annoying to wait what seems like forever to save a file to a USB disk drive, but operating slowly is much better than writing incorrect data to the disk file.

For devices that require both correct data transmission and timely delivery, USB uses *interrupt* transfers. Despite their name, interrupt transfers do not involve interrupts on the computer system. Instead, the USB protocol marks interrupt transfers as high-priority events. The host polls all devices on the USB, but the devices do not interrupt the host when they have data available. A peripheral device using the interrupt transfer type may request how often the host polls it, choosing an interval from 1 to 255 milliseconds.³

In order to guarantee correct and timely delivery of interrupt transmissions between a host and a peripheral device, the USB host controller stack must reserve a portion of the USB bandwidth whenever an application opens a device for interrupt transmission. For example, if a particular device wants to be serviced every millisecond and needs to transmit 16 bytes per packet, the USB host controller stack must reserve a little bit more than 128Kbps (kilobits per second) of bandwidth (16 bytes $\times$ 8 bits per byte $\times$ 1,000 packets per second) from the total bandwidth available. You need to reserve a little bit more than this, because there's some protocol overhead on the bus as well—at least 10 to 20 percent, but it could be more depending upon how the USB stack is written.

2. In theory, you could use control transmissions to pass data between the peripheral and the host, but very few devices do so.

3. The host may legally poll the device more often than the device requests. The specified polling time is a *minimum* polling interval.

Because there's a limited amount of bandwidth available on the USB, and because interrupt transmissions consume a fixed amount of that bandwidth whenever you open a device for use, you can't have an arbitrary number of interrupt transmissions active at any one time. Once the USB bandwidth (minus the 10 percent that USB reserves for control transmissions) is consumed, the stack refuses to activate any new interrupt transmissions.

Interrupt transmission packets are between 4 and 64 bytes long, though most of the time they fall into the low end of this range. Larger packets would prevent the system from guaranteeing the desired polling frequency.

Many devices use interrupt transmissions to notify the host CPU that some data is available, and then the host uses a bulk transmission to actually read the data from the device. If the amount of data to transmit between the host and the peripheral is small enough, the peripheral may transmit the data as part of the interrupt's data payload to avoid a second transmission. Keyboards, mice, joysticks, and similar devices typically transmit their data this way. Disk drives, scanners, and other such devices use interrupt transmissions to notify the host that data is available and then use bulk transfers to move the data around.

Isochronous (or *iso*) transfers are the fourth transfer type that USB supports. Like interrupt transfers, iso transfers require a timely delivery. Like bulk transfers, they generally involve larger data packets. However, unlike the other three transfer types, they do not guarantee correct delivery between the host and the peripheral device. Timely delivery is so important for iso transfers that if a packet arrives late, it might as well not arrive at all. Peripheral devices such as audio input (microphones) and output (speakers) and video cameras use iso transmissions. If you lose a packet, or if a packet is transmitted incorrectly between the peripheral and host, you'll get a momentary glitch on the video display or in the audio signal, but such problems are not disastrous as long as they don't occur too frequently.

Like interrupt transfers, iso transfers consume USB bandwidth. Whenever you open a connection to an iso USB peripheral device, that device requests a certain amount of bandwidth. If the bandwidth is available, the USB host controller stack reserves it for the device until the application is finished with the device. If sufficient bandwidth is not available, the USB stack notifies the application that it cannot use the desired device until the user stops using other iso and interrupt devices to free up some bandwidth.

13.3.4 USB-C

USB originally competed with FireWire for mindshare among peripheral developers. Early on, FireWire was a much higher-performing interface and protocol. However, with the advent of USB-2 and, especially, USB-3, FireWire became less attractive. During this time, Apple worked with Intel to create a new external peripheral bus protocol—Thunderbolt. Thunderbolt totally smoked USB on performance. The race was on again, this time between USB and Thunderbolt. However, Intel (which promoted both USB and Thunderbolt) decided to merge the two standards into one: USB-C. USB-C is actually a Thunderbolt 3 hardware interface that happens to carry USB, PCI,

SCSI, and other protocols over the serial bus. Now, you don't really have to decide—USB-C (or Thunderbolt-3) is the interface of choice.

13.3.5 USB Device Drivers

Most OSes that provide a USB stack support dynamic loading and unloading of USB device drivers, known as *client drivers* in USB terminology. Whenever you attach a USB device to the USB, the host system gets a signal telling it that the *bus topology* has changed (that is, there's a new device on the USB). The host controller scans for the new device, a process known as *enumeration*, and then reads some configuration information from the peripheral. Among other things, this configuration information tells the USB stack the type of the device, the manufacturer, and model information. The USB host stack uses this information to determine which device driver to load into memory. If the USB stack can't find a suitable driver, it generally opens up a dialog box requesting help from the user; if the user can't provide the path to an appropriate driver, the system simply ignores the new device. Similarly, when the user unplugs a device, the USB stack unloads the appropriate device driver from memory if it's not also being used for some other device.

To simplify device driver implementation for many common devices, such as keyboards, disk drives, mice, and joysticks, the USB standard defines certain device classes. Peripheral manufacturers who create devices that adhere to one of these standardized device classes don't have to supply a device driver with their equipment. Instead, the class drivers that come with the USB host controller stack provide the only interface necessary. Examples of class drivers include HID (Human Interface Devices, such as keyboards, mice, and joysticks), STORAGE (disk, CD, and tape drives), COMMUNICATIONS (modems and serial converters), AUDIO (speakers, microphones, and telephony equipment), and PRINTERS. Peripheral manufacturers can always opt to supply their own specialized features that add bells and whistles to their product, but a customer will often get basic functionality from some existing class driver by simply plugging in the device without installing a device driver specifically for it.

13.4 For More Information

Axelson, Jan. *USB Complete: The Developer's Guide*. 4th ed. Madison, WI: Lakeview Publishing, 2009.

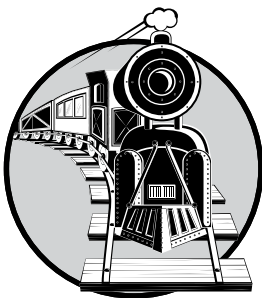
Field, Gary, Peter M. Ridge et al. *The Book of SCSI*. 2nd ed. San Francisco: No Starch Press, 2000.

NOTE

For the USB, FireWire, and TCP/IP (network) protocol stacks, you'll find considerable information online. For example, <http://www.usb.org/> contains all the technical specifications for the USB protocol as well as programming information for various common USB host controller chip sets. You'll also find plenty of online code resources, such as complete source code from Linux for TCP/IP and USB host controller stacks.

14

MASS STORAGE DEVICES AND FILESYSTEMS



The most prevalent I/O device on modern computers is probably the mass storage device. Whereas some PCs don't have a display (they're operated *headlessly*), or even a keyboard or mouse (they're accessed remotely), almost every computer system recognizable as a PC has a mass storage device of some sort. This chapter will focus on the types of mass storage devices—hard drives, floppy disks, tape drives, flash drives, solid state drives, and more—as well as the special filesystem format they use to organize the data they store.

14.1 Disk Drives

Almost all modern computer systems include some sort of disk drive unit to provide online mass storage. At one time, certain workstation vendors produced *diskless workstations*, but the relentless drop in price and increasing storage space of fixed (aka “hard”) disk and solid-state drive (SSD)

units have all but obliterated the diskless computer system. Disk drives are so ubiquitous in modern systems that most people take them for granted. However, it's dangerous for a programmer to take a disk drive for granted. Software constantly interacts with the disk drive as a medium for application file storage, so it's very important to understand how disk drives operate if you want to write efficient code.

14.1.1 Floppy Disk Drives

Floppy disks have all but disappeared from today's PCs. Their limited storage capacity (typically 1.44MB) is far too small for modern applications and the data they produce. It's hard to believe that at the beginning of the PC revolution a 143KB (that's *kilobytes*, not megabytes or gigabytes) floppy drive was considered a high-ticket item. However, floppy disk drives have failed to keep up with technological advances in the computer industry. Therefore, we won't consider them further in this chapter.

14.1.2 Hard Drives

The fixed disk drive, more commonly known as the hard drive, is the most common mass storage device in use today (though, as of 2020, SSDs are rapidly replacing hard drives). The modern hard drive is truly an engineering marvel. Between 1982 and 2020, the capacity of a single drive unit has increased over 2,400,000-fold, from 5MB to over 16TB (terabytes). At the same time, the minimum price for a new unit has dropped from \$2,500 (US) to below \$50. No other component in the computer system has enjoyed such a radical increase in capacity and performance along with a comparable drop in price. (Semiconductor RAM probably comes in second: paying the 1982 price today would get you about 40,000 times the capacity.)

While hard drives were decreasing in price and increasing in capacity, they were also becoming faster. In the early 1980s, a hard-drive subsystem was doing well to transfer 1MBps between the drive and the CPU's memory; modern hard drives can transfer more than 2,500MBps.¹ While this increase in performance isn't as great as that of memory or CPUs, keep in mind that disk drives are mechanical units on which the laws of physics place greater limitations. In some cases, the dropping cost of hard drives has allowed system designers to improve their performance by using disk arrays (see "RAID Systems" on page 388 for details). By using certain hard-disk subsystems like disk arrays, you could achieve 2500MBps (or better) transfer rates, though it's not especially cheap to do so.

Hard drives are so named because their data is stored on a small, rigid disk that is usually made out of aluminum or glass and is coated with a magnetic material. Floppy disks, in contrast, store their information on a thin piece of flexible Mylar plastic.

1. Well, at least when connected in a high-performance RAID system.

In disk-drive terminology, the small aluminum or glass disk is known as a *platter*. Each platter has two surfaces, front and back (or top and bottom), both of which have the magnetic coating. During operation, the hard-drive unit spins this platter at a particular speed, which these days is usually 3,600; 5,400; 7,200; 10,000; or 15,000 revolutions per minute (RPM). Generally, though not always, the faster the platter spins, the faster the data is read from the disk and the higher the data transfer rate between the disk and the system. The smaller disk drives in laptop computers typically spin at much slower speeds, like 2,000 or 4,000 RPM, to conserve battery life and generate less heat.

A hard-disk subsystem contains two main active components: the disk platter(s) and the read/write head. The read/write head, when held stationary, floats above concentric circles, or *tracks*, on the disk surface. Each track is broken up into a sequence of sections known as *sectors* or *blocks*. The actual number of sectors varies by drive design, but a typical hard drive has between 32 and 128 sectors per track (see Figure 14-1). Each sector typically holds between 256 and 4,096 bytes of data. Many disk-drive units let the OS choose between several different sector sizes, the most common being 512 bytes and 4,096 bytes.

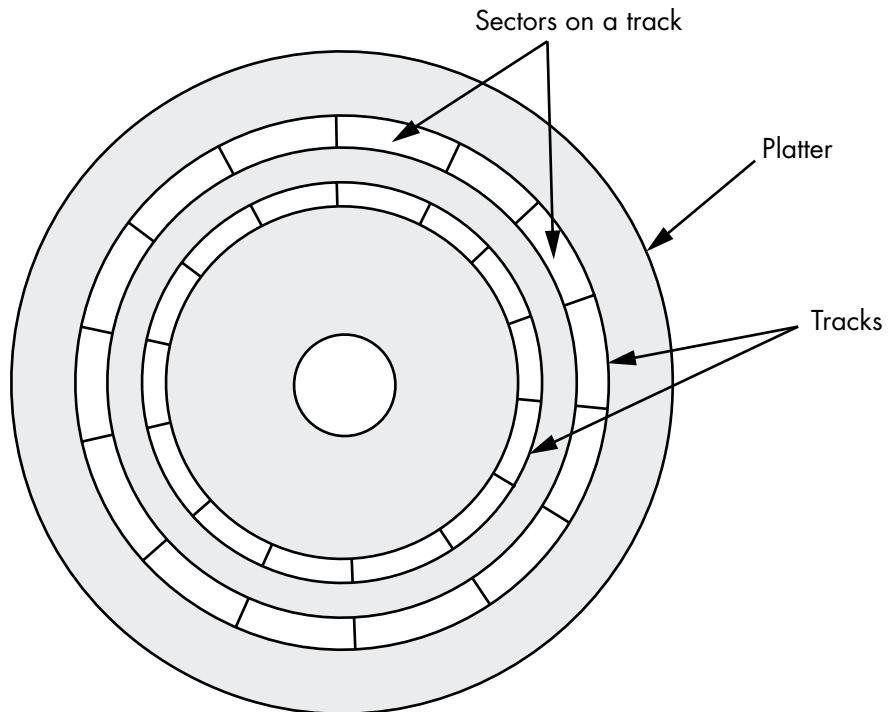


Figure 14-1: Tracks and sectors on a hard-disk platter

The disk drive records data when the read/write head sends a series of electrical pulses to the platter, which translates them into magnetic pulses that the platter's magnetic surface retains. The frequency at which the disk controller can record these pulses is limited by the quality of the electronics, the read/write head design, and the quality of the magnetic surface.

The magnetic medium is capable of recording two adjacent bits on its disk surface and then differentiating between them during a later read operation. However, as you record bits closer and closer together, it becomes increasingly difficult to differentiate between them in the magnetic domain. *Bit density* is a measure of how closely a particular hard disk can pack data into its tracks—the higher the bit density, the more data you can squeeze onto a single track. However, recovering densely packed data requires faster and more expensive electronics.

The bit density has a big impact on the performance of the drive. If the drive's platters are rotating at a fixed number of RPM, then the higher bit density, the more bits will rotate underneath the read/write head over a certain duration. Larger disk drives tend to be faster than smaller disk drives because they employ a higher bit density.

By moving the disk's read/write head in a roughly linear path from the center of the disk platter to the outside edge, the system can position a single read/write head over any one of several thousand tracks. Yet the use of only one read/write head means that it will take a fair amount of time to move the head among the disk's many tracks. Indeed, two of the most cited hard-disk performance parameters are the read/write head's average seek time and track-to-track seek time.

The *average seek time* is half the amount of time it takes to move the read/write head from the edge of the disk to the center, or vice versa. A typical high-performance disk drive has an average seek time between 5 and 10 milliseconds. On the other hand, its *track-to-track seek time*—that is, the amount of time it takes to move the disk head from one track to the next—is on the order of 1 or 2 milliseconds. From these numbers, you can see that the acceleration and deceleration of the read/write head consumes a much greater percentage of the track-to-track seek time than of the average seek time. It takes only 20 times longer to traverse 1,000 tracks than it does to move to the next track. And because moving the read/write heads from one track to the next is usually the most common operation, the track-to-track seek time is probably a better indication of the disk's performance. Regardless of which metric you use, however, keep in mind that moving the disk's read/write head is one of the most expensive operations you can do on a disk drive, so it's something you want to minimize.

Because most hard-drive subsystems record data on both sides of a disk platter, there are two read/write heads associated with each platter—one for the top and one for the bottom. And because most hard drives incorporate multiple platters in their disk assembly in order to increase storage capacity (see Figure 14-2), a typical drive has multiple pairs of read/write heads.

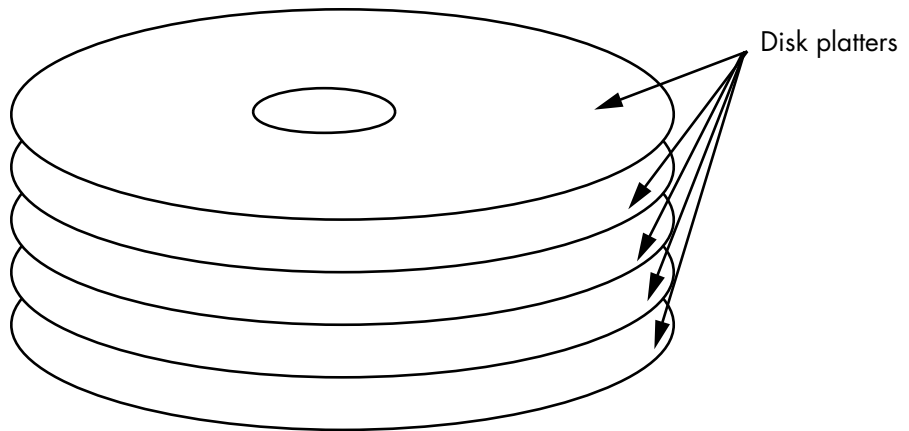


Figure 14-2: Multiple-platter hard-disk assembly

The various read/write heads are physically connected to the same actuator. Therefore, each head sits above the same track on its respective platter, and all the heads move across the disk surfaces as a unit. The set of all tracks over which the read/write heads are currently sitting is known as a *cylinder* (see Figure 14-3).

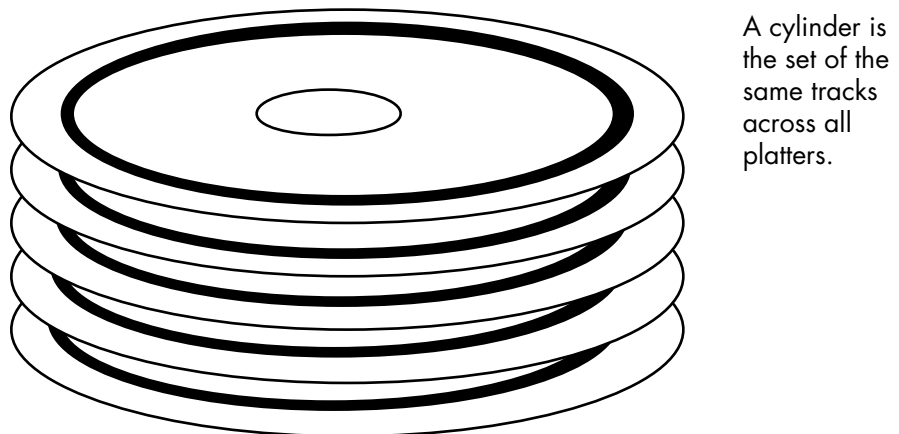


Figure 14-3: A hard-disk cylinder

Although using multiple heads and platters increases the cost of a hard-disk drive, it also improves the performance. The performance boost occurs when data the system needs isn't located on the current track. In a hard-disk subsystem with only one platter, the read/write head would need to move to another track to locate the data. But in a subsystem with multiple platters, the next block of data to read is usually located within the same cylinder. And because the hard-disk controller can quickly switch between read/write heads electronically, doubling the number of platters in a disk

subsystem nearly doubles the disk unit's track-to-track seek performance because it winds up doing half the number of seek operations. Of course, increasing the number of platters also increases the unit's capacity, which is another reason why high-capacity drives are often higher-performance drives as well.

With older disk drives, when the system wants to read a particular sector from a particular track on one of the platters, it commands the disk to position the read/write head over the appropriate track, and the disk drive then waits for the desired sector to rotate underneath. But by the time the head settles down, there's a chance that the desired sector has just passed under the head, in which case the disk has to wait for almost one complete rotation before it can read the data. On average, the desired sector appears halfway across the disk. If the disk is rotating at 7,200 RPM (120 revolutions per second), it requires 8.333 milliseconds for one complete rotation of the platter. Typically, 4.2 milliseconds will pass before the sector rotates underneath the head. This delay is known as the *average rotational latency* of the drive, and it is usually equal to the time needed for one rotation, divided by 2.

To see how average rotational latency can be a problem, consider that an OS usually manipulates disk data in sector-sized chunks. For example, when reading data from a disk file, the OS typically requests that the disk subsystem read a sector of data and return that data. Upon receiving the data, the OS processes it and then very likely makes a request for additional data from the disk. But what happens when this second request is for data located on the next sector of the current track? Unfortunately, while the OS is processing the first sector's data, the disk platters are still moving underneath the read/write heads. If the OS wants to read the next sector on the disk's surface but doesn't notify the drive immediately after reading the first sector, the second sector will rotate underneath the read/write head. When this happens, the OS will have to wait for almost a complete disk rotation before it can read the second sector. This is known as *blowing revs* (revolutions). If the OS (or application) is constantly blowing revs when reading data from a file, filesystem performance suffers dramatically. In early "single-tasking" OSes running on slower machines, blowing revs was an unpleasant fact of life. If a track had 64 sectors, it would often take 64 revolutions of the disk in order to read all the data on a single track.

To combat this problem, the disk-formatting routines for older drives allow the user to interleave sectors. *Interleaving* is the process of spreading out sectors within a track so that logically adjacent sectors are not physically adjacent on the disk surface (see Figure 14-4).

The advantage of interleaving sectors is that once the OS reads a sector, it will take a full sector's rotation time before the logically adjacent sector moves under the read/write head. This gives the OS time to do some processing and to issue a new disk I/O request before the desired sector moves underneath the head. However, in modern multitasking OSes, it's difficult to guarantee that an application will gain control of the CPU so that it can respond before the next logical sector moves under the head, so interleaving isn't very effective.

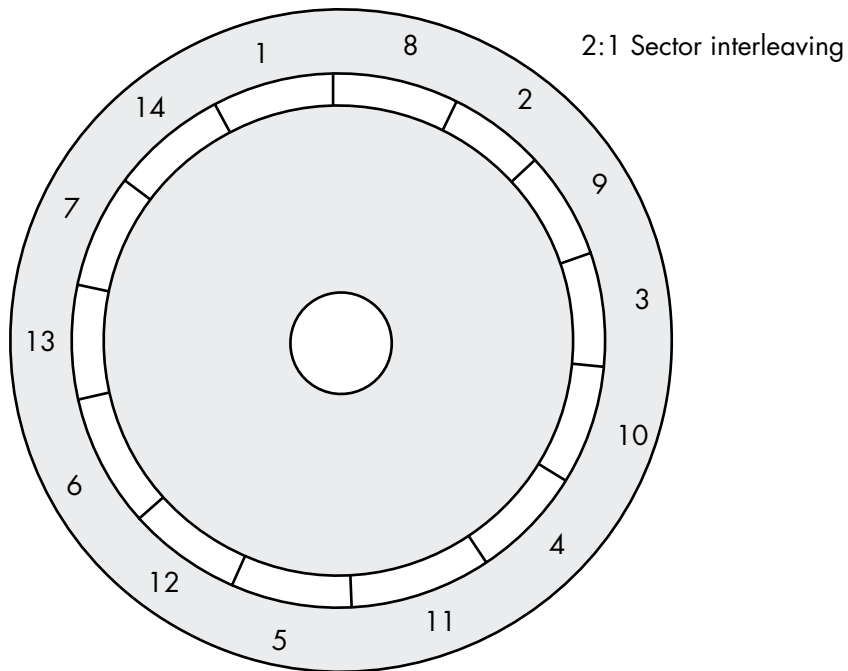


Figure 14-4: Interleaving sectors

To solve this problem, as well as improve disk performance in general, most modern disk drives include memory on the disk controller that allows it to read data from an entire track in one disk revolution. Once it caches the track data in memory, the controller can communicate disk read/write operations at RAM speed rather than at disk rotation speeds, which can dramatically improve performance. Reading the first sector from a track still exhibits rotational latency, but once the disk controller reads the entire track, the latency is all but eliminated for that track.

A typical track may have 64 sectors of 512 bytes each, for a total of 32KB per track. Because newer disks usually have between 8MB and 512MB of on-controller memory, the controller can buffer hundreds of tracks in its memory. Therefore, the disk controller cache improves not only the performance of disk read/write operations on a single track, but also overall disk performance. Note that the disk controller cache speeds up read operations *and* write operations. For example, the CPU can often write data to the disk controller's cache memory within a few microseconds and then return to normal data processing while the disk controller moves the disk read/write heads into position. When the disk heads are finally in position at the appropriate track, the controller can write the data from the cache to the disk surface.

From an application designer's perspective, advances in disk subsystem design have reduced the need to understand how disk-drive geometries (track and sector layouts) and disk-controller hardware affect the application's performance. Despite these attempts to make the hardware transparent to the application, though, software engineers wanting to write great

code must always remain cognizant of the disk drive's underlying operation. For example, it's valuable to know that sequential file operations are usually much faster than random-access operations because sequential operations require fewer head seeks. Also, if you know that a disk controller has an on-board cache, you can write file data in smaller blocks, doing other processing between the block operations, to give the hardware time to write the data to the disk surface. Though the techniques early programmers used to maximize disk performance don't apply to modern hardware, by understanding how disks operate and how they store their data, you can avoid various pitfalls that produce slow code.

14.1.3 RAID Systems

Because a modern disk drive typically has between 8 and 16 heads, you might wonder if you could improve performance by simultaneously reading or writing data on multiple heads. While this is certainly possible, it really didn't happen until SATA and larger disk caches came along. But there's yet another way to improve disk drive performance using parallel read and write operations—the *redundant array of inexpensive disks* (RAID) configuration.

The RAID concept is quite simple: you connect multiple hard-disk drives to a special host controller card (sometimes known as an *adapter*), which simultaneously reads and writes the various disk drives. By hooking up two disk drives to a RAID controller card, you can read and write data about twice as fast as you could with a single disk drive. By hooking up four disk drives, you can improve average performance by almost a factor of 4.

RAID controllers support different configurations depending on the purpose of the disk subsystem. So-called *RAID 0* subsystems use multiple disk drives simply to increase the data transfer rate. If you connect two 150GB disk drives to a RAID controller, you'll produce the equivalent of a 300GB disk subsystem with double the data transfer rate. This is a typical configuration for personal RAID systems—those systems that are not installed on a file server.

Many high-end file-server systems are *RAID 1* (and higher) subsystems that store multiple copies of the data across the multiple disk drives, rather than increasing the data transfer rate between the system and the disk drive. In such configurations, should one disk fail, a copy of the data is still available on another disk drive. Some even higher-level RAID subsystems combine four or more disk drives to increase the data transfer rate and provide redundant data storage. This type of configuration usually appears on high-end, high-availability file server systems.

Modern RAID system configurations can be categorized as follows:

RAID 0 Interleaves data across all disks to increase performance (at the expense of reliability). This is known as *striping*. Requires a minimum of two disks.

RAID 1 Replicates data on pairs of drives to increase reliability (at the cost of performance; also cuts in half the total amount of storage available). Allows failure of at least one drive without data loss (depending on the drives that fail, could support two or more drive failures).

Requires an even number of drives, with a minimum of two disks. This is known as *mirroring*.

RAID 5 Stores parity information on the drives. Faster than RAID 1, slower than RAID 0. Allows failure of one drive without data loss. Requires a minimum of three drives. At three drives, 66 percent of the total storage is available for data; any drives you add beyond three increase data storage by the size of the added drive.

RAID 6 Stores duplicate parity information across the drives. Faster than RAID 1, slower than RAID 0 and 5. Allows failure of two drives without data loss. Requires a minimum of four drives. At four drives, half the total storage is available for data, but any drives you add beyond four increase system storage by the size of the added drive.

RAID 10 Combination of RAID 1 + RAID 0. Minimum four drives; expansion has to be in pairs of drives. Interleaved (striped) data across drives to speed up performance, plus redundant storage on pairs of drives for reliability. Faster than RAID 1 (but slower than RAID 0).

RAID 50, 60 Combination of RAID 5 + RAID 0 or RAID 6 + RAID 0.

There are other RAID combinations (like 2, 3, and 4), but most are obsolete and you won't find them in use in modern systems.

RAID systems enable you to dramatically increase disk subsystem performance without having to purchase exotic and expensive mass storage solutions. Though a software engineer can't assume that every computer system in the world has a fast RAID subsystem available, for those applications that demand the absolute highest-performance storage subsystem, RAID (possibly using SSDs) could be a solution.

14.1.4 Optical Drives

An optical drive uses a laser beam and a special photosensitive medium to record and play back digital data. Optical drives have a few advantages over hard-disk subsystems that use magnetic media:

- They are more shock resistant, so banging the disk drive around during operation won't destroy the drive unit as easily as it would a hard disk.
- The medium is usually removable, allowing you to maintain an almost unlimited amount of offline or near-line storage.
- They're fairly high-capacity (though modern USB memory sticks and SD cards have greater capacities).

At one time, optical storage systems appeared to be the wave of the future because they offered very high storage capacity in a small space. Unfortunately, they have fallen out of favor in all but a few niche markets because they also have several drawbacks:

- While their read performance is okay, their write speed is very slow—an order of magnitude slower than a hard drive and only a few times faster than a *floptical* (older combined magnetic/optical floppy) drive.

- Although the optical medium is far more robust than the magnetic medium, the magnetic medium in a hard drive is usually sealed away from dirt, humidity, and abrasion. In contrast, optical media is easily accessible to someone who really wants to do damage to the disk's surface.
- Seek times for optical-disk subsystems are much slower than for magnetic disks.
- Optical disks have limited storage capacity, currently less than about 128GB (Blu-ray).

Ultimately, the low price and increasing capacity of USB flash drives killed off optical drives for personal computer use.

One area where optical-disk subsystems are still in use, however, is in *near-line storage subsystems*, which typically use a robotic jukebox to manage hundreds or thousands of optical disks. Although you could argue that a rack of high-capacity hard-disk drives would provide a more space-efficient storage solution, it would consume far more power, generate far more heat, and require a more sophisticated interface than an optical jukebox, which usually has only a single optical-drive unit and a robotic disk-selection mechanism. For archival storage, where the server system rarely needs access to any particular piece of data in the storage subsystem, a jukebox system is a very cost-effective solution.

If you wind up writing software that manipulates files on an optical-drive subsystem, the most important thing to remember is that read access is much faster than write access. You should try to use the optical system as a “read-mostly” device and avoid writing data as much as possible to the device. You should also avoid random access on an optical disk's surface, as seek times are very slow.

CD, DVD, and Blu-ray drives are also optical drives. However, because of their widespread use, and their sufficiently different organization and performance when compared with standard optical drives, they warrant a separate discussion.

14.1.5 CD, DVD, and Blu-ray Drives

CD-ROM was the first optical drive subsystem to gain wide acceptance in the personal computer market. CD-ROM disks were based on the audio CD digital recording standard, and they provided a large amount of storage (650MB) when compared to hard-disk-drive storage capacities at the time (typically 100MB). As time passed, of course, this relationship reversed. Still, CD-ROMs became the preferred distribution vehicle for most commercial applications, completely replacing the floppy-disk medium for this purpose.

Although the CD-ROM format is a very inexpensive distribution medium in large quantities, often costing only a few cents per disk, it's not appropriate for small production runs. The problem is that it typically costs several hundreds or thousands of dollars to produce a disk master (from which the run of CD-ROMs are made), meaning that CD-ROM is usually cost-effective only when the quantity of disks being produced is at least in the thousands.

The solution was a new CD medium, CD-Recordable (CD-R), which allowed the production of one-off CD-ROMs. CD-R uses a write-once optical disk technology, known euphemistically as *WORM* (write-once, read-many). When first introduced, CD-R disks cost about \$10 to \$15. However, once the drives reached critical mass and media manufacturers began producing blank CD-R disks in huge quantities, their bulk retail price fell to about \$0.25. As a result, CD-R made it possible to distribute a fair amount of data in small quantities.

One obvious drawback to CD-R is the “write-once” limitation. To overcome it, the CD-Rewriteable (CD-RW) drive and medium were created. CD-RW, as its name suggests, supports both reading and writing. Unlike with optical disks, however, you can’t simply rewrite a single sector on CD-RW. Instead, to rewrite the data on a CD-RW disk, you must first erase the whole disk.

Although the 650MB of storage on a CD seemed like a gargantuan amount when CDs were first introduced, the old maxim that data and programs expand to fill up all available space certainly held true. Though CDs were ultimately expanded to 700MB, various games (with embedded video), large databases, developer documentation, programmer development systems, clip art, stock photographs, and even regular applications reached the point where a single CD was woefully inadequate. The DVD-ROM (and later, DVD-R, DVD-RW, DVD+RW, and DVD-RAM) disk reduced this problem by offering between 3GB and 17GB of storage on a single disk. Except for the DVD-RAM format, you can view the DVD formats as faster, higher-capacity versions of the CD formats. There are some clear technical differences between the two, but most of them are transparent to the software. Today, Blu-ray optical discs deliver up to 128GB of storage (Blu-ray BDXL). However, electronic distribution via the internet has largely replaced physical media, so Blu-ray discs have never become as popular as distribution or storage media.

The CD and DVD formats were created for reading data in a continuous stream—*streaming* data—from the storage medium. The track-to-track head movement time required to read data stored on a hard disk creates a big gap in the streaming sequence, which is unacceptable for audio and video applications. CDs and DVDs record information on a single, very long track that forms a spiral across the surface of the whole disk. Thus, the CD or DVD player can continuously read the data simply by moving the laser beam along the disk’s single spiral track at a constant rate.

Although having a single track is great for streaming data, it does make it a bit more difficult to locate a specific sector on the disk. The CD or DVD drive can only approximate a sector’s position by mechanically positioning the laser beam to some point on the disk. Next, it must actually read data from the disk surface to determine where the laser is positioned, and then do some fine-tuning to locate the desired sector. As a result, searching for a specific sector on a CD or DVD disk can take an order of magnitude longer than searching for a specific sector on a hard disk.

The most important thing to remember for a programmer writing code that interacts with CD or DVD media is that random access is verboten.

These media were designed for sequential streaming access, and seeking data on such media will hinder your application performance. If you're using these disks to deliver your application and its data to the end user, you should have the user copy the data to a hard disk before use if high-performance random access is necessary.

14.2 Tape Drives

Tape drives were also popular mass storage devices. Traditionally, PC owners used tape drives to back up data stored on hard-disk drives back in the days when hard drives were much smaller. For many years, tape storage was far more cost-effective than hard-disk storage on a cost-per-megabyte basis. Indeed, at one time there was an order of magnitude difference in cost per megabyte between tape storage and magnetic disk storage. And because tape drives held more data than most hard-disk drives, they were more space-efficient too.

However, because of competition and technological advances in the hard-disk-drive marketplace, tapes have lost these advantages. Hard-disk drives now exceed 16TB in storage, and the optimum price point for hard disks is about \$0.25 per gigabyte. Tape storage today costs far more per megabyte than hard-disk storage. Plus, only a few tape technologies allow you to store 250GB on a single tape, and those that do (such as Digital Linear Tape, or DLT) are extremely expensive. It's not surprising that tape drives are seeing less and less use these days in home PCs and are typically found only in larger file server machines. Linear Tape-Open (LTO) drives extend the capacity to around 12TB (expected to increase to around 200TB in the future). Nevertheless, today a typical LTO-8 tape costs almost \$130 (US), about half the price per megabyte of a hard drive.

Back in the days of mainframes, application programs interacted with tape drives in much the same way that today's applications interact with hard-disk drives. A tape drive, however, is not an efficient random-access device. That is, although software can read a random set of blocks from a tape, it cannot do so with acceptable performance. Of course, in the days when most applications ran on mainframes, applications generally were not interactive, and CPUs were much slower; thus, the standard for "acceptable performance" was different.

In a tape drive, the read/write head is fixed, and the tape transport mechanism moves the tape past the head linearly, from the beginning of the tape to the end, or vice versa. If the beginning of the tape is currently positioned over the read/write head and you want to read data at the end of the tape, you have to move the entire tape past the head to get to the desired data. This can be very slow, requiring tens or even hundreds of seconds, depending on the length and format of the tape. Compare this with the tens of milliseconds it takes to reposition a hard disk's read/write head (or the negligible time it takes to get data from an SSD). Therefore, to perform well on a tape drive, software must be written to account for the

limitations of a sequential access device. In particular, data should be read or written sequentially on a tape.

Originally, data was written to tapes in blocks (much like sectors on a hard disk), and the drives were designed to allow quasi-random access to the tape's blocks. If you've ever watched old movies that used the reel-to-reel drives, with the reels constantly stopping, starting, stopping, reversing, stopping, and continuing, you've seen "random access" in action. Such tape drives were very expensive because they required powerful motors, finely tooled tape-path mechanisms, and so on. As hard drives became larger and less expensive, applications stopped using tape as a data manipulation medium and used it only for offline storage (to back up data from hard disks).

Because sequential data access on tape does not require the heavy-duty mechanics of the original tape drives, tape-drive manufacturers sought to make a lower-cost product suitable for sequential access only. Their solution was the *streaming tape drive*, which was designed to keep the data constantly moving from the CPU to the tape, or vice versa. For example, while backing up the data from a hard disk to tape, a streaming tape drive treats the data like a video or audio recording and just lets the tape run, constantly writing the data from the hard disk to the tape. Because of the way streaming tape drives work, very few applications deal directly with the tape unit. Today, it's very rare for anything other than a tape backup utility program, run by the system administrator, to access the tape hardware.

14.3 Flash Storage

An interesting storage medium that has become popular because of its compact form factor² is flash storage. The flash medium is actually a semiconductor device, based on *electrically erasable programmable read-only memory* (EEPROM) technology, which, despite its name, is both readable and writable. Unlike regular semiconductor memory, flash storage is *nonvolatile*, meaning it maintains its data even when disconnected from power. Like other semiconductor technologies, flash storage is purely electronic and doesn't require any motors or other electromechanical devices for proper operation. Therefore, flash storage devices are more reliable and shock resistant, and they use far less power than mechanical storage solutions such as disk drives. This makes flash storage especially valuable in portable battery-powered devices like cell phones, tablets, laptop computers, electronic cameras, MP3 playback devices, and recorders.

Flash storage modules now provide in excess of 1TB of storage, and their optimal price point is about \$0.15 (US) per gigabyte. This makes them comparable, per bit, to hard-disk storage.

Flash devices are sold in many different form factors. OEMs (original equipment manufacturers) can buy flash storage devices that look like other semiconductor chips and mount them directly on their circuit boards. However, most flash memory devices sold today are built into one

2. In this context, "form factor" means shape and size.

of several standard forms, including SDHC cards, CompactFlash cards, smart-memory modules, memory sticks, USB/flash modules, or SSDs. For example, you might remove a CompactFlash card from your camera, insert it into a special CompactFlash card reader on your PC, and access your photographs just as you would files on a disk drive.

Memory in a flash storage module is organized in blocks of bytes, not unlike sectors on a hard disk. In contrast to regular semiconductor memory or RAM, however, you can't write individual bytes in a flash storage module. Although you can generally *read* an individual byte from a flash storage device, to write to a particular byte you must first erase the entire block on which it resides. The block size varies by device, but most OSes treat these flash blocks like a disk sector for the purposes of reading and writing. Although the basic flash storage device itself could connect directly to the CPU's memory bus, most common flash storage packages (such as CompactFlash cards and memory sticks) contain electronics that simulate a hard-disk interface, and you access the flash device just as you would a hard-disk drive.

One interesting aspect to flash memory devices, and EEPROM devices in general, is that they have a limited write lifetime. That is, you can write to a particular memory cell in a flash memory module only a certain number of times before that cell begins to have problems retaining the information. This was a big concern in early EEPROM/flash devices, because the average number of write cycles before failures began occurring was around 10,000. That is, if some software wrote to the same memory block 10,000 times in a row, the EEPROM/flash device would probably develop a bad memory cell in that block, effectively rendering the entire chip useless. On the other hand, if the software wrote just once to 10,000 separate blocks, the device could still take 9,999 more writes to each memory cell. Therefore, the OSes of these early devices would try to spread out write operations across the entire device to minimize damage. Although modern flash devices still exhibit this problem, technological advances have reduced it almost to the point where we can ignore it. A modern flash memory cell supports an average of about a million write cycles before it will go bad. Furthermore, today's OSes simply mark bad flash blocks, the same way they mark bad sectors on a disk, and will skip a block once they determine that it has gone bad.

Being electronic, flash devices do not exhibit rotational latency times at all, and they don't exhibit much in the way of seek times either. There's a tiny amount of time needed to write an address to a flash memory module, but it's nothing compared to the head seek times on a hard disk. Despite this, flash memory is generally nowhere near as fast as typical RAM. Reading data from a flash device itself usually takes microseconds (rather than nanoseconds), and the interface between the flash memory device and the system may require additional time to set up a data transfer. In addition, it's common to interface a flash storage module to a PC using a USB flash reader device, and this can further reduce the average read time per byte to hundreds of microseconds.

Write performance is even worse. To write a block of data to flash, you must write the data, read it back, compare it to the original data, and rewrite it if they don't match. This process can take several tens or even hundreds of milliseconds.

As a result, flash memory modules are generally quite a bit slower than high-performance hard-disk subsystems. However, thanks mainly to demand from high-end digital camera users who want to be able to snap as many pictures as possible in a short time, technological advances are boosting their performance. Though flash memory performance probably won't catch up with hard-disk performance any time soon, you can expect it to continue improving over time.

14.4 RAM Disks

Another interesting mass storage device is the RAM disk, a semiconductor solution that treats a large block of the computer system's memory as though it were a disk drive, simulating blocks and sectors using memory arrays. The advantage of memory-based disks is that they are very high performance. RAM disks don't suffer from the time delays associated with head seek time and rotational latency that you find on hard, optical, and floppy drives. Their interface to the CPU is also much faster, so data transfer times are very short, often running at the maximum bus speed. It's hard to imagine a faster storage technology than a RAM disk.

RAM disks, however, have two disadvantages: cost and volatility. The cost per byte of storage in a RAM disk system is very high. Indeed, byte for byte, semiconductor storage is as much as 10,000 times more expensive than magnetic hard-disk storage. As a result, RAM disks usually have low storage capacities, typically no more than several gigabytes. And RAM disks are volatile—they lose their memory unless they are powered at all times. This generally means that semiconductor disks are great for storing temporary files and files you'll copy back to some permanent storage device before shutting down the system. They are not particularly well suited for maintaining important information over long periods of time.

14.5 Solid-State Drives

Modern high-performance PCs use solid-state drives (SSDs). SSDs use flash memory (like USB sticks) with a high-performance interface to the system. But SSDs aren't simply USB flash drives in different clothing. USB flash drives are designed for low cost per bit—except for certain camera applications (particularly 4K and 8K camcorders), speed is secondary to cost and capacity. A typical USB flash drive, for example, is quite a bit slower than a mediocre hard drive. SSDs, on the other hand, must be fast. Because of their solid-state design, they're typically an order of magnitude faster than rotating magnetic media. With a RAID configuration, SSDs can actually achieve the performance limits of SATA interfaces.

As this was being written, SSDs cost between 4 and 16 times as much as high-capacity hard drives (8TB drives and 1TB SSDs both cost about \$100 US). However, the price-per-gigabyte gap has been closing. SSDs are rapidly replacing rotating magnetic drives, and rotating magnetic media will likely be relegated to the trash bin of history (much like tape drives). Before that point, why would anyone pay more for an SSD?

SSDs typically use a different underlying technology to store data and provide a much faster electronic interface to the PC. This is why an SSD tends to be much more expensive than a USB flash drive. That's also why SSDs can achieve 2,500MBps data transfer rates, while high-quality memory cards are capable of only around 100MBps (and USB flash/thumb drives are even worse).

From a programmer's perspective, one of the big advantages of SSDs is that you no longer have to worry about seek times and other latency issues. SSDs tend to be true(r) random-access devices (at least when compared with hard drives). Accessing data at the beginning of the drive and then at the end takes only a little longer than accessing any pair of data elements elsewhere on the SSD.

There are a couple of disadvantages to SSDs, though. First of all, their write performance is usually much slower than their read performance (though writing to an SSD is still much faster than writing to a hard drive). Fortunately, data is read far more often than it is written, but this is something to consider when you're working on software that writes data to a SSD. The second drawback is that SSDs wear out after a while. Writing to the same location over and over again will eventually cause the associated memory cell(s) to fail. Fortunately, modern OSes work around these failures. However, when you write applications that continuously overwrite file data, keep this issue in mind.

14.6 Hybrid Drives

Most modern hard drives contain an on-board RAM cache (to hold entire tracks of data to eliminate rotational latency, for example). Hybrid drives, such as Apple's older Fusion Drive, combine a small SSD with a large hard drive—typically a 32GB to 128GB SSD and a 2TB magnetic disk, in Apple's case. Frequently accessed data stays in the SSD cache, and is swapped out to the hard drive when space is needed for new data. This works the same way as caching in main memory, boosting the system performance to near-SSD speeds for data that is accessed regularly.

14.7 Filesystems on Mass Storage Devices

Very few applications access mass storage devices directly. That is, applications do not generally read and write tracks, sectors, or blocks on a mass storage device; instead, they open, read, write, and otherwise manipulate *files* on it. The OS's *file manager* abstracts away the physical

configuration of the underlying storage device and provides a convenient storage facility for multiple independent files on a single device.

On the earliest computer systems, applications were responsible for tracking the physical location of data on a mass storage device, because there was no file manager available to do so. They were able to maximize their performance by carefully considering the layout of data on the disk. For example, they could manually interleave data across various sectors on a track to give the CPU time to process it between reading and writing those sectors on the track. Such software was often many times faster than comparable software using a generic file manager. Later, when file managers were commonly available, some application authors still managed their files on a storage device for performance reasons. This was especially true back in the days of floppy disks, when low-level software written to manipulate data at the track and sector level often ran 10 times faster than the same application using a file manager system.

In theory, today's software could benefit from this approach as well, but in practice you rarely see this kind of low-level disk access in modern applications, for several reasons. First, writing software that manipulates a mass storage device at such a low level locks you into using that particular device. That is, if your software manipulates a disk with 48 sectors per track, 12 tracks per cylinder, and 768 cylinders per drive, that software will not work optimally (if at all) on a drive with a different sector, track, and cylinder layout. Second, accessing the drive at a low level makes it difficult to share the device among different applications, something that can be especially costly on a multitasking system that may have several applications sharing the device at once. For example, if you've laid out your data on various sectors on a track to coordinate computation time with sector access, your work is lost when the OS interrupts your program and gives some other application its time slice—time you were counting on to do any necessary computations prior to the next data sector rotating under the read/write head. Third, some of the features of modern mass storage devices, such as on-board caching controllers and SCSI interfaces that present a storage device as a sequence of blocks rather than as something with a given track and sector geometry, eliminate any advantage that low-level software might have had. Fourth, modern OSes typically contain file buffering and block caching algorithms that provide good filesystem performance, obviating the need to operate at such a low level. Finally, low-level disk access is very complex, and writing such software is difficult.

14.7.1 Sequential Filesystems

The earliest file manager systems stored files sequentially on the disk's surface. That is, if each sector/block on the disk held 512 bytes and a file was 32KB long, that file would consume 64 consecutive sectors/blocks on the disk's surface. To access that file at some future time, the file manager only needed to know the file's starting block number and the number of blocks it occupied. The filesystem had to maintain these two pieces of information

somewhere in nonvolatile storage. The obvious place was on the storage media itself, in a data structure known as the *directory*—an array of values starting at a specific disk location that the OS can reference when an application requests a particular file. The file manager can search through the directory for the file’s name and extract its starting block and length. With this information, the filesystem can provide the application with access to the file’s data.

One advantage of the sequential filesystem is that it is very fast. The OS can read or write a single file’s data very rapidly if the file is stored in sequential blocks on the disk’s surface. But a sequential file organization has some serious problems, too. The biggest and most obvious drawback is that you can’t extend the size of a file once the file manager places another file at the next block on the disk. Disk fragmentation is another big problem. As applications create and delete many small and medium files, the disk fills up with short sequences of unused sectors that, individually, are too small for most files. On sequential filesystems, disks often had free space sufficient to hold some data, but they couldn’t use it because it was scattered in small pieces all over the disk’s surface. To solve this problem, users had to run disk compaction programs to coalesce all the free sectors and move them to the end of the disk by physically rearranging files on its surface. Another solution was to copy files from one full disk to another empty disk, collecting the many small, unused sectors together. Obviously, this was extra work that the user had to do—work that the OS should have been doing.

The sequential file storage scheme really falls apart when used with multitasking OSes. If two applications attempt to write file data to the disk concurrently, the filesystem must place the starting block of the second application’s file beyond the last block required by the first application’s file. As the OS has no way of determining how large the files can grow, each application has to tell the OS the maximum length of the file when the application first opens it. Unfortunately, many applications cannot determine in advance how much space they’ll need for their files, so they have to guess the size of the file when opening it. If the estimated file size is too small, either the program has to abort with a “file full” error, or the application has to create a larger file, copy the old data from the “full” file to the new file, and then delete the old file. As you can imagine, this is horribly inefficient and definitely not great code.

To avoid such performance problems, many applications grossly overestimate the amount of space they need for their files. As a result, they wind up wasting disk space when the files don’t actually use all the data allocated to them, a form of *internal* fragmentation. Furthermore, if applications truncate their files when closing them, the resulting free sections returned to the OS tend to fragment the disk into the small, unusable blocks of free space described previously, a problem known as *external* fragmentation. For these reasons, sequential storage on the disk has been replaced by more sophisticated storage management schemes in modern OSes.

14.7.2 Efficient File Allocation Strategies

Most modern file allocation strategies allow files to be stored across arbitrary blocks on the disk. Because the filesystem can now place bytes of the file in any free block on the disk, the problems of external fragmentation and the limitation on file size are all but eliminated. As long as there's at least one free block on the disk, you can expand the size of any file.

However, with this flexibility comes some added complexity. In a sequential filesystem, it was easy to locate free space on the disk; by simply noting the starting block numbers and sizes of the files in a directory, the filesystem could locate a free block large enough to satisfy the current disk allocation request, if one was available. But with files stored across arbitrary blocks, scanning the directory and noting which blocks a file uses is far too expensive to compute, so the filesystem has to keep track of the free and used blocks. Most modern OSes use one of three data structures—a set, a table (array), or a list—to keep track of which sectors are free and which are not. Each scheme has its advantages and disadvantages.

14.7.2.1 Free-Space Bitmaps

The free-space bitmap scheme uses a set data structure to maintain a set of free blocks on the disk drive. If a block is a member of that set, the file manager can remove it whenever it needs another block for a file. Because set membership is a Boolean relationship (a block is either in the set or it's not), it takes exactly 1 bit to specify the set membership of each block.

Typically, a file manager reserves a certain section of the disk to hold a bitmap that specifies which blocks on the disk are free. The bitmap consumes some integral number of blocks on the disk, with each block consumed representing a specific number of other blocks on the disk, which we can calculate by multiplying the block size (in bytes) by 8 (bits per byte). For example, if the OS uses 4,096-byte blocks on the disk, a bitmap consisting of a single block can track up to 32,768 other blocks on the disk.

The disadvantage of the bitmap scheme is that as disks get large, so does the bitmap. For example, on a 120GB drive with 4,096-byte blocks, the bitmap will be almost 4MB long. While this is a small percentage of the total disk capacity, accessing a single bit in a bitmap this large can be clumsy. To find a free block, the OS has to do a linear search through this 4MB bitmap. Even if you keep the bitmap in system memory (which is a bit expensive, considering that you have to do it for each drive), searching through it every time you need a free sector is an expensive proposition. As a result, you don't see this scheme used much on larger disk drives.

One advantage (and also a disadvantage) of the bitmap scheme is that the file manager uses it only to keep track of the free space on the disk, not which sectors belong to a given file. As a result, if the free-space bitmap is damaged somehow, nothing is permanently lost; you can easily reconstruct it by searching through all the disk directories and computing which sectors are being used by the files in those directories (the remaining sectors, obviously, are the free ones). Although this process is somewhat time-consuming, it's nice to have the option if disaster strikes.

14.7.2.2 File Allocation Tables

Another way to track disk-sector usage is with a table of sector pointers, or a *file allocation table (FAT)*. This scheme is widely used. Cementing its popularity, this is also the default file allocation scheme used on most USB flash drives. An interesting facet of the FAT structure is that it combines both free-space management and file-sector allocation management into the same data structure, ultimately saving space when compared to the bitmap scheme, which uses separate data structures for each. Furthermore, unlike the bitmap scheme, FAT doesn't require an inefficient linear search to find the next available free sector.

The FAT is really nothing more than an array of self-relative pointers (that is, indexes into itself), setting aside one pointer for each sector/block on the storage device. When a disk is initialized, the first several blocks on its surface are reserved for objects like the root directory and the FAT itself, and the remaining blocks on the disk are free. Somewhere in the root directory is a free-space pointer that specifies the next available free block on the disk. Assuming the free-space pointer initially contains the value 64, implying that the next free block is block 64, the FAT entries at indexes 64, 65, 66, and so on, would contain the following values, assuming there are n blocks on the disk, numbered from 0 to $n - 1$:

FAT index	FAT entry value
...	...
64	65
65	66
66	67
67	68
...	...
$n - 2$	$n - 1$
$n - 1$	0

The entry at block 64 tells you the next available free block on the disk, 65. Moving on to entry 65, you'll find the value of the next available free block on the disk, 66. The last entry in the FAT contains a 0 (block 0 contains meta-information for the entire disk partition and is never available).

Whenever an application needs one or more blocks to hold some new data on the disk's surface, the file manager grabs the free-space pointer value and then continues going through the FAT entries for however many blocks are required to store the new data. For example, if each block is 4,096 bytes long and the current application is attempting to write 8,000 bytes to a file, the file manager will need to remove two blocks from the free-block list, following these steps:

1. Get the value of the free-space pointer.
2. Save the value of the free-space pointer in order to determine the first free sector.

3. Continue going through the FAT entries for the number of blocks required to store the application's data.
4. Extract the FAT entry value of the last block where the application needs to store its data, and set the free-space pointer to this value.
5. Store a 0 over the FAT entry value of the last block that the application uses, marking the end to the list of blocks that the application needs.
6. Return the original (as it was prior to these steps) value of the free-space pointer into the FAT as the pointer to the list of blocks that are now allocated for the application.

After the block allocation in our earlier example, the application has blocks 64 and 65 at its disposal, the free-space pointer contains 66, and the FAT looks like this:

FAT index	FAT entry value
...	...
64	65
65	0
66	67
67	68
...	...
$n - 2$	$n - 1$
$n - 1$	0

This is not to imply that entries in the FAT *always* contain the index of the next entry in the table. As the file manager allocates and deallocates storage for files on the disk, these numbers tend to become scrambled. For example, if an application returns block 64 to the free list but holds on to block 65, the free-space pointer would contain the value 64, and the FAT would have the following values:

FAT index	FAT entry value
...	...
64	66
65	0
66	67
67	68
...	...
$n - 2$	$n - 1$
$n - 1$	0

As noted earlier, one advantage of the FAT data structure is that it combines both free-space management and file allocation management

into a single data structure. This means that each file doesn't have to carry around a list of the blocks its data occupies. Instead, it needs only a single pointer value specifying an index into the FAT where the first block of the file's data can be found. You can find the remaining blocks containing the file's data by stepping through the FAT.

One important advantage of the FAT scheme over the set (bitmap) scheme is that once the disk using a FAT filesystem is full, it doesn't maintain information about which blocks are free. In contrast, the bitmap scheme consumes space on the disk to track free blocks even when there are none available. The FAT scheme replaces the entries originally used to track free blocks with the file-block pointers. When the disk is full, the values that originally maintained the free-block list are no longer consuming disk space because they're all now tracking blocks in files. In this case, the free-space pointer contains 0 (to denote an empty free-space list) and all the FAT entries contain chains of block indexes for file data.

However, the FAT scheme does have a couple of disadvantages. First, unlike the bitmap in a set scheme filesystem, the table in a FAT filesystem represents a single point of failure. If the FAT is somehow destroyed, it can be very difficult to repair the disk and recover files; losing some free space on a disk is a problem, but losing track of where your files are on the disk is a *major* problem. Furthermore, because the disk head tends to spend more time in the FAT area of a storage device than in any other single area on the disk, the FAT is the most likely part of a hard disk to be damaged by a head crash, and the most likely part of a floppy or optical drive to exhibit excessive wear. This is a sufficiently big concern that some FAT filesystems provide an option to maintain an extra copy of the FAT on the disk.

Another problem with the FAT is that it's usually located at a fixed place on the disk, typically at some low block number. In order to determine which block or blocks to read for a particular file, the disk heads must move to the FAT, so if the FAT is at the beginning of the disk, they'll constantly be traveling to and from the FAT across large distances. This massive head movement not only is slow but tends to wear out the mechanical parts of the disk drive sooner. In newer versions of Microsoft OSes, the FAT-32 scheme eliminates part of this problem by allowing the FAT to be positioned somewhere other than the beginning of the disk, though still at a fixed location. Application file I/O performance can be quite low with a FAT filesystem unless the OS caches the FAT in main memory, which can be dangerous if the system crashes, because you could lose track of all file data whose FAT entries have not been written to disk.

The FAT scheme is also inefficient for doing random access on a file. To read from offset m to offset n in a file, the file manager must divide n by the block size to obtain the block offset into the file containing the byte at offset n , divide m by the block size to obtain its block offset, and then sequentially search through the FAT chain between these two blocks to find the sector(s) containing the desired data. This linear search can be expensive if the file is a large database with many thousands of blocks between the current block position and the desired block position.

Yet another problem with the FAT filesystem, though this one is rather esoteric, is that it doesn't support sparse files. That is, you cannot write to byte 0 and byte 1,000,000 of a file without also allocating every byte of data between those two points on the disk surface. Some non-FAT file managers allocate only the blocks where an application has written data. For example, if an application writes data only to bytes 0 and 1,000,000 of a file, the file manager allocates only two blocks for the file. If the application attempts to read a block that hasn't been previously allocated (for example, if the application in the current example attempts to read the byte at offset 500,000 without first writing to that location), the file manager simply returns 0s for the read operation without actually using any space on the disk. But because of how a FAT is organized, you can't create sparse files on the disk.

14.7.2.3 Lists of Blocks

To overcome the limitations of the FAT filesystem, advanced OSes—such as Windows NT/2000/XP/7/8/10, macOS (APFS), and various flavors of Unix—use a list-of-blocks scheme instead. Indeed, the list scheme enjoys all the advantages of a FAT system (such as efficient, nonlinear free-block location, and efficient storage of the free-block list), and it solves many of FAT's problems.

The list scheme begins by setting aside several blocks on the disk for the purpose of keeping (generally) 32- or 64-bit pointers to each free block on the disk. If each block on the disk holds 4,096 bytes, a block can hold 1,024 (or 512) pointers. Dividing the number of blocks on the disk by 1,024 (512) determines the number of blocks the free-block list will initially consume. As you'll soon see, the system uses these blocks to store data once the disk fills up, so there's no storage overhead associated with the blocks consumed by the free-block list.

If a block in the free-block list contains 1,024 pointers (the following examples all assume 32-bit pointers), then the first 1,023 pointers contain the block numbers of free blocks on the disk. The file manager maintains two pointers on the disk: one that holds the block number of the current block containing free-block pointers, and one that holds an index into that current block. Whenever the filesystem needs a free block, it obtains the index for one from the free-block list by using these two pointers. Then the file manager increments the index into the free-block list to the next available entry in the list. When the index increments to 1,023 (the 1,024th item in the free-block list), instead of using the pointer entry value at that index to locate a free block, the file manager uses it as the address of the next block containing a list of free-block pointers on the disk, and it uses the current block, containing a now-empty list of block pointers, as the free block. This is how the file manager reuses the blocks originally designated to hold the free-block list, rather than reusing the pointers in the free-block list to keep track of the blocks belonging to a given file, as FAT does. Once the file manager uses up all the free-block pointers in a given block, it uses that block for actual file data.

Unlike the FAT, the list scheme does not merge the free-block list and the file list into the same data structure. Instead, a separate data structure for each file holds the list of blocks associated with that file. Under typical Unix and Linux filesystems, the directory entry for the file actually holds the first 8 to 16 entries in the list (see Figure 14-5). This allows the OS to track small files (up to 32KB or 64KB) without having to allocate any extra space on the disk.

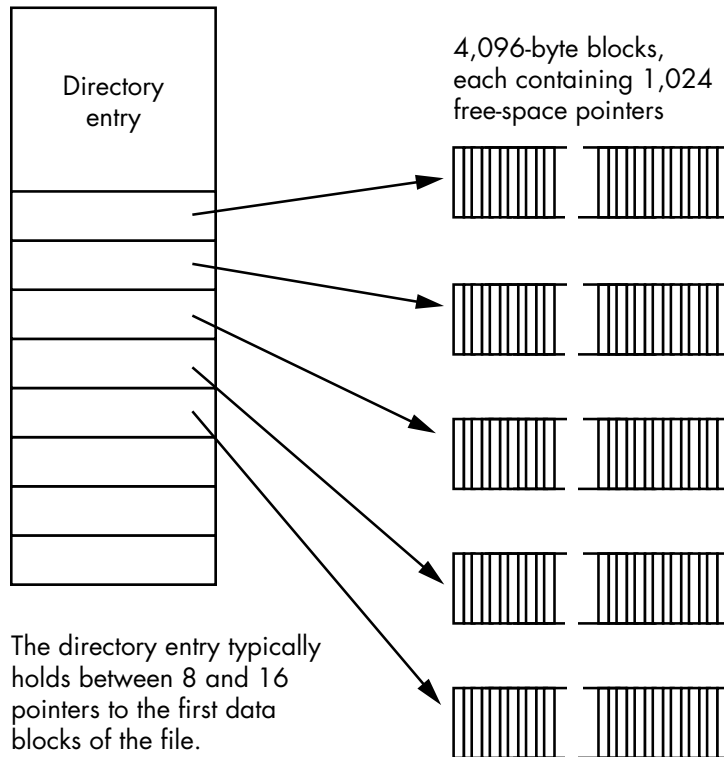


Figure 14-5: Block list for small files

Research on various flavors of Unix suggests that the vast majority of files are small, and embedding several pointers into the directory entry provides an efficient way to access small files. Of course, as time passes, the average file size seems to increase. But as it turns out, block sizes tend to increase as well. When the research was first done, the typical block size was 512 bytes, but today it's 4,096 bytes. During that time, then, average file sizes could have increased by a factor of 8 without, on average, requiring any extra space in the directory entries.

For medium files, up to about 4MB, the OS will allocate a single block with 1,024 pointers to the blocks that store the file's data. The OS continues to use the pointers found in the directory entry for the first few blocks of the file, and then it uses a block on the disk to hold the next group of block pointers. Generally, the last pointer in the directory entry holds the location of this block (see Figure 14-6).

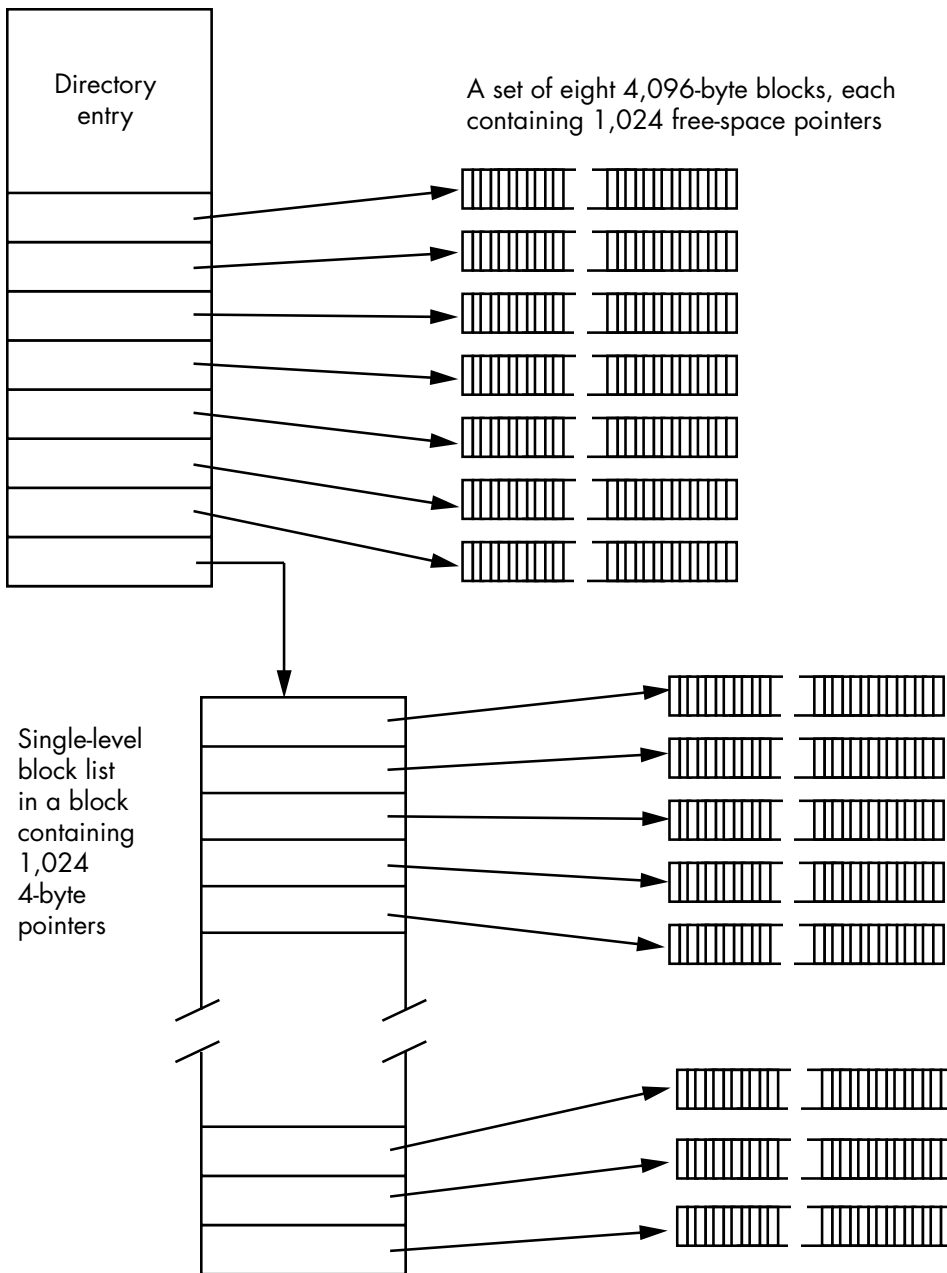


Figure 14-6: Block list for medium files

For files larger than about 4MB, the filesystem switches to a three-tiered block scheme, which works for file sizes up to 4GB. In this scheme, the last pointer in the directory entry stores the location of a block of 1,024 pointers, and each pointer in this block holds the location of an additional block of 1,024 pointers, with each pointer in *this* block storing the location of a block that contains actual file data. See Figure 14-7 for the details.

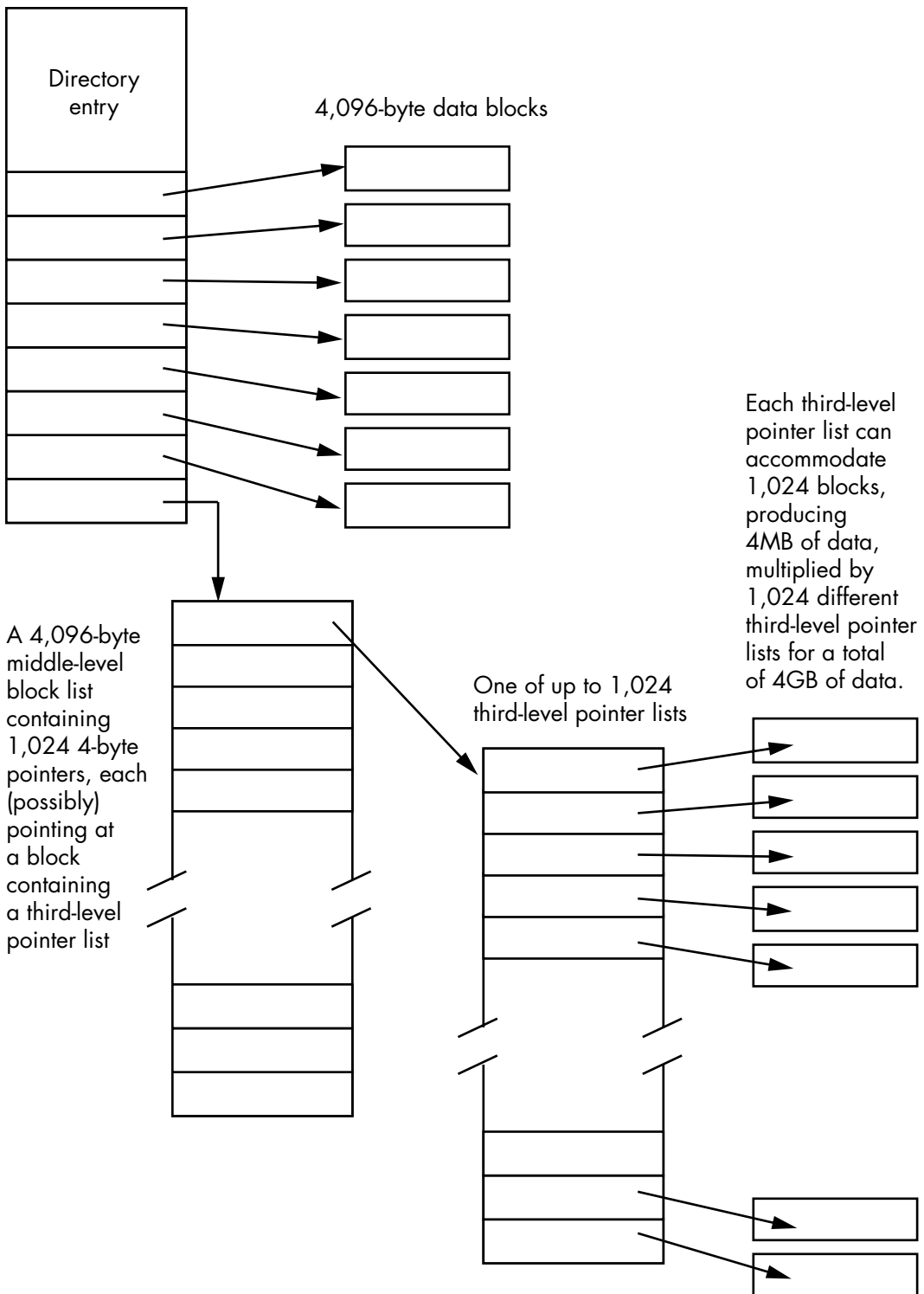


Figure 14-7: Three-level block list for large files (up to 4GB)

One advantage to this tree structure is that it readily supports sparse files: an application can write to block 0 and block 100 of a file without having to allocate data blocks for every block between those two points. By placing a special block pointer value (typically 0) in the intervening entries in the block list, the OS can determine whether a block isn't present in the file. Should an application attempt to read a missing block in the file, the OS can simply return all 0s for the empty block. Of course, once the application writes data to a block that hadn't been previously allocated, the OS must copy the data to the disk and fill in the appropriate block pointer in the block list.

As disks became larger, the 4GB file limit imposed by this scheme began to create some problems for certain applications, such as video editors, large database applications, and web servers. One could easily extend this scheme 1,000 times—to 4TB—by adding another level to the block-list tree. The only problem with this approach is that the more levels of indirection you have, the slower random file access becomes, because the OS may have to read several blocks from the disk in order to get a single block of data. (When it has one level, it makes sense to cache the block-pointer list in memory, but with two and three levels, it's impractical to do this for every file). Another way to extend the maximum size 4GB at a time is to use multiple pointers to second-tier file blocks (for example, have all or most of the original 8 to 16 pointers in the directory point at second-tier block-list entries rather than directly at file data blocks). Although there's no current convention for extending beyond three levels, rest assured that as the need arises, OS designers will develop schemes for accessing large files efficiently. For example, 64-bit OSes can use 64-bit pointers, rather than 32-bit pointers, and eliminate the 4GB limitation.

14.8 Writing Software That Manipulates Data on a Mass Storage Device

Understanding how different mass storage devices behave is important if you want to write high-performance software that manipulates files on these devices. Although modern OSes attempt to isolate applications from the physical realities of mass storage, an OS can only do so much for you. Furthermore, because an OS can't predict how your particular application will access files on a mass storage device, it can't tailor file access optimizations to your application; instead, its optimizations are geared toward applications exhibiting *typical* file access patterns. The less typical your application's file I/O is, then, the less likely you'll get the best performance out of the system. In this section, we'll look at how you can coordinate your file access activities with the OS to achieve the best performance.

14.8.1 File Access Performance

Although disk drives and most other mass storage devices are often thought of as "random access" devices, the fact is that mass storage access is usually more efficient when done sequentially. Sequential access on a disk drive is relatively efficient because the OS can move the read/write head one track

at a time (assuming the file appears in sequential blocks on the disk). This is much faster than accessing one block on the disk, moving the read/write head to some other track, accessing another block, moving the head again, and so on. Therefore, you should avoid random file access in an application if at all possible.

You should also try to read or write large blocks of data on each file access rather than reading or writing small amounts more frequently. There are two reasons for this. First, OS calls are not fast, so if you make half as many calls by reading or writing twice as much data on each access, the application will often run twice as fast. Second, the OS must read or write whole disk blocks. If your block size is 4,096 bytes, but you just write 2,000 bytes to some block and then seek to some other position in the file outside that block, the OS will actually have to read the entire 4,096-byte block from the disk, merge in the 2,000 bytes, and then finally write the entire 4,096 bytes back to the disk. Contrast this with a write operation that writes a full 4,096 bytes—in this case, the OS wouldn't have to read the data from the disk first; it would only have to write the block. Writing full blocks improves disk access performance by a factor of 2, because writing partial blocks requires the OS to first read the block, merge the data, and then write the block; writing whole blocks renders the read operation unnecessary. Even if your application doesn't write data in increments that are even multiples of the disk's block size, writing large blocks improves performance. If you write 16,000 bytes to a file in one write operation, the OS will still have to write the last block of those 16,000 bytes using a read-merge-write operation, but it will write the first three blocks using only write operations.

If you start with a relatively empty disk, the OS generally attempts to write the data for new files in sequential blocks. This organization is probably most efficient for future file access. However, as the system's users create and delete files on the disk, the blocks of data for individual files may be distributed nonsequentially. In a very bad case, the OS may wind up allocating a few blocks here and a few blocks there all across the disk's surface. As a result, even sequential file access can behave like slow random file access. As discussed previously, this kind of file fragmentation can dramatically decrease filesystem performance. Unfortunately, there's no way for an application to determine if its file data is fragmented across the disk surface and, even if it could, there's little it could do about the situation. Although there are utilities available to *defragment* the blocks on the disk's surface, an application generally can't request their execution (and “defragger” utilities are quite slow anyway).

Although applications rarely get the opportunity to defragment their data files during normal program execution, there are some things you can do to reduce the probability of your data files becoming fragmented. The best advice you can follow is to always write file data in large chunks. Indeed, if you can write the whole file in a single write operation, do so. In addition to speeding up access to the OS, writing large amounts of data tends to result in the allocation of sequential blocks. When you write small blocks of data to the disk, other applications in a multitasking environment

could also be writing to the disk concurrently. In this case, the OS may interleave the block allocation requests for the files being written by several different applications, making it unlikely that a particular file's data will be written sequentially. It is important to try to write a file's data in sequential blocks, even if you plan to access portions of that data randomly, since searching for random records in a file written sequentially generally requires far less head movement than searching for random records in a file whose blocks are scattered all over.

If you're going to create a file and then access its blocks of data repeatedly, whether randomly or sequentially, try to preallocate the blocks on the disk. If you know, for example, that your file's data will not exceed 1MB, you could write a block of one million 0s to the disk before your application starts manipulating the file. By doing so, you help ensure that the OS will write your file to sequential blocks on the disk. Though you pay an initial price to write all those 0s (an operation you wouldn't normally do, presumably), the savings in read/write head-seek times could easily make up for it. This scheme is especially useful if an application is reading or writing two or more files concurrently (which would almost guarantee the interleaving of the blocks for the various files).

14.8.2 Synchronous and Asynchronous I/O

Because most mass storage devices are mechanical, and, therefore, subject to mechanical delays, applications that use them extensively have to wait for them to complete read/write operations. Most disk I/O operations are *synchronous*, meaning that an application that makes a call to the OS must wait until that I/O request is complete before continuing subsequent operations.

This is why most modern OSes also provide an *asynchronous I/O* capability, in which the OS begins the application's request and then returns control to the application without waiting for the I/O operation to complete. While the I/O operation proceeds, the application promises not to do anything with the data buffer specified for it. However, the application can do computation and schedule additional I/O operations, because the OS will notify it when the original request completes. This is especially useful when you're accessing files on multiple disk drives in the system, which is usually possible only with SCSI and other high-end drives.

14.8.3 The Implications of I/O Type

Another important consideration for writing software that manipulates mass storage devices is the type of I/O you're performing. *Binary I/O* is usually faster than *formatted text I/O*, because of the format of the data written to disk. For example, suppose you have an array of 16 integer values that you want to write to a file. To achieve this, you could use either of the following two C/C++ code sequences:

```
FILE *f;  
int array[16];  
...
```

```
// Sequence #1:

fwrite( f, array, 16 * sizeof( int ));
. . .
// Sequence #2:

for( i=0; i < 16; ++i )
    fprintf( f, "%d ", array[i] );
```

The second sequence looks like it would run slower than the first because it uses a loop, rather than a single call, to step through each element of the array. But although the extra execution overhead of the loop does have a small negative impact on the execution time of the write operation, this efficiency loss is minor compared to the real problem with the second sequence. Whereas the first code sequence writes out a 64-byte memory image consisting of 16 32-bit integers to the disk, the second code sequence converts each of the 16 integers to a string of characters and then writes each string to the disk. This integer-to-string conversion process is relatively slow. Furthermore, the `fprintf()` function has to interpret the format string ("`%d`") at runtime, which incurs an additional delay.

The advantage of formatted I/O is that the resulting file is both human-readable and easily read by other applications. However, if you're using a file to hold data that is of interest only to your application, a more efficient approach might be to write the data as a memory image.

14.8.4 Memory-Mapped Files

Memory-mapped files use the OS's virtual memory capabilities to map memory addresses in the application space directly to blocks on the disk. Modern OSes have highly optimized virtual memory subsystems, so piggy-backing file I/O on top of the virtual memory subsystem results in very efficient file access. Furthermore, memory-mapped file access is easy. When you open a memory-mapped file, the OS returns a memory pointer to some block of memory. By simply accessing the memory locations referenced by this pointer, just as you would any other in-memory data structure, you can access the file's data. This makes file access almost trivial, while often improving file manipulation performance, especially when file access is random.

One of the reasons that memory-mapped files are so much more efficient than regular files is that the OS reads the list of blocks belonging to memory-mapped files only once. It then sets up the system's memory management tables to point at each block belonging to the file. After opening the file, the OS rarely has to read any file metadata from the disk, which greatly reduces superfluous disk access during random file access. It also improves sequential file access, though to a lesser degree. The OS doesn't constantly have to copy data between the disk, internal OS buffers, and application data buffers.

Memory-mapped file access does have some disadvantages. First, you can't map gigantic files entirely into memory, at least on older PCs and OSes that have a 32-bit address bus and set aside a maximum of 4GB per

application. Generally, it isn't practical to use a memory-mapped access scheme for files larger than 256MB, though this has changed as more CPUs with 64-bit addressing capabilities have become available. It's also not a good idea to use memory-mapped files when an application is already using most of the RAM physically present in the system. Fortunately, these two situations are not typical, so they don't limit the use of memory-mapped files much.

A more common and significant issue is that when you first create a memory-mapped file, you have to tell the OS the file's maximum size. If it's impossible to determine the file's final size, you'll have to overestimate it and then truncate the file when you close it. Unfortunately, this wastes system memory while the file is open. Memory-mapped files work well when you're manipulating files in read-only mode or simply reading and writing data in an existing file without extending the file's size. Fortunately, you can always create a file using traditional file access mechanisms and then use memory-mapped file I/O to access the file later.

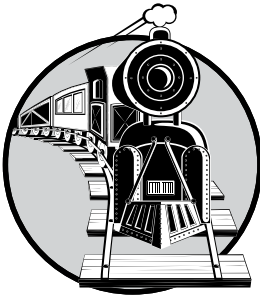
Finally, almost every OS does memory-mapped file access differently, so it's unlikely that memory-mapped file I/O code will be portable between OSes. Nevertheless, the code to open and close memory-mapped files is quite short, and it's easy enough to provide multiple copies of the code for the various OSes you need to support. Of course, actually accessing the file's data consists of simple memory accesses, and that's independent of the OS. For more information on memory-mapped files, consult your OS's API reference. Given their convenience and performance, you should seriously consider using memory-mapped files whenever possible in your applications.

14.9 For More Information

Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 8th ed. Hoboken, NJ: John Wiley & Sons, 2009.

15

MISCELLANEOUS INPUT AND OUTPUT DEVICES



Although mass storage devices are, arguably, the most common peripheral in modern computer systems, there are many other widely used devices, such as communication ports (serial and parallel), keyboards and mice, and sound cards. These peripherals will be the focus of this chapter.

15.1 Exploring Specific PC Peripheral Devices

In some respects, it's dangerous to discuss real devices on modern PCs because the traditional ("legacy") devices have all but disappeared from PC designs. As manufacturers introduce new PCs, they are removing many of the legacy, easy-to-program peripherals like parallel and serial ports, and replacing them with complex peripherals like USB and Thunderbolt. Although a detailed discussion on programming these newer peripheral

devices is beyond the scope of this book, you need to understand their behavior in order to write great code that accesses them.

NOTE

Because of the nature of the peripheral devices discussed in the rest of this chapter, the information presented applies only to IBM-compatible PCs. There simply isn't enough space in this book to cover how particular I/O devices behave on different systems. Other systems support similar I/O devices, but their hardware interfaces may differ from what's described here. Nevertheless, the general principles still apply.

15.1.1 The Keyboard

The original IBM PC's keyboard is a computer system in its own right. Buried inside the keyboard's case is an 8042 microcontroller chip that constantly scans the switches on the keyboard to see if any keys are being pressed. This processing occurs in parallel with the normal activities of the PC, and even though the PC's 80x86 is busy with other things, the keyboard never misses a keystroke.

A typical keystroke starts with the user pressing a key on the keyboard. This closes an electrical contact in a switch, which the keyboard's microcontroller can sense. Unfortunately, mechanical switches do not always close perfectly clean. Often, the contacts bounce off one another several times before coming to rest with a solid connection. To a microcontroller chip that is reading the switch constantly, these bouncing contacts look like a very quick series of keypresses and releases. If the microcontroller registers these as multiple keystrokes, it can result in a phenomenon known as *keybounce*, a problem common to many cheap and old keyboards. Even on the most expensive and newest keyboards, keybounce can be a problem if you look at the switch a million times a second, because mechanical switches simply cannot settle down that quickly. A typical inexpensive key will settle down within 5 milliseconds, so if the keyboard-scanning software polls the key less often than this, the controller will effectively miss the keybounce. The practice of limiting how often the keyboard is scanned in order to eliminate keybounce is known as *debouncing*. Typical keyboard controllers scan the keyboard once every 10 to 25 milliseconds; any less than this may produce bouncy keys, and any more may result in lost keystrokes (by very fast typists).

The keyboard controller must not generate a new key code sequence every time it scans the keyboard and finds a key held down. The user may hold a key down for many tens or hundreds of milliseconds before releasing it, and we don't want this to register as multiple keystrokes. Instead, the keyboard controller should generate a single key code value when the key goes from the up position to the down position (a *down key* operation). In addition, modern keyboards provide an *autorepeat* capability that engages once the user has held down a key for a given time period (usually about half a second), and it treats the held key as a sequence of keystrokes as long as the user continues to hold the key down. However, even these autorepeat keystrokes are regulated to allow only about 10 keystrokes per second rather than the number of times per second the keyboard controller scans all the switches on the keyboard.

Upon detecting a down keystroke, the microcontroller sends a keyboard *scan code* to the PC. The scan code is *not* related to the ASCII code for that key; it is an arbitrary value IBM chose when the PC's keyboard was first developed. The PC keyboard actually generates *two* scan codes for every key pressed. It generates a *down code* when a key is pressed down and an *up code* when the key is released. If the user holds the key down long enough for the autorepeat operation to begin, the keyboard controller sends a sequence of down codes until the key is released, at which point the keyboard controller sends a single up code.

The 8042 microcontroller chip transmits these scan codes to the PC, where they are processed by an *interrupt service routine (ISR)* for the keyboard. Having separate up and down codes is important because certain keys (like SHIFT, CTRL, and ALT) are meaningful only when held down. By generating up codes for all the keys, the keyboard ensures that the keyboard ISR knows which keys are pressed while the user is holding down one of these *modifier* keys. Exactly what the system does with these scan codes depends on the OS, but usually the OS's keyboard device driver will translate the scan code sequence into an appropriate ASCII code or some other notation that applications can work with.

Today, almost all PC keyboards interface via the USB port, and they probably use a more modern microcontroller than the 8042 found in the original IBM PC keyboard, but otherwise their behavior is exactly the same.

15.1.2 The Standard PC Parallel Port

The original IBM PC design provided support for three parallel printer ports (which IBM designated *LPT1:*, *LPT2:*, and *LPT3:*). With laser and inkjet printers still a few years in the future, IBM probably envisioned machines that could support a standard dot matrix printer, a daisy wheel printer, and maybe some other auxiliary type of printer for different purposes. IBM almost certainly didn't anticipate the widespread use of parallel ports, or it probably would have designed them differently. At their prime, the PC's parallel port controlled keyboards, disk drives, tape drives, SCSI adapters, Ethernet and other network adapters, joystick adapters, auxiliary keypad devices, other miscellaneous devices, and, oh yes, printers.

Today, the parallel port is largely absent in systems because of connector size and performance problems. Nevertheless, it remains an interesting device. It's one of the few interfaces that hobbyists can use to connect the PC to simple devices they've built themselves. Therefore, learning to program the parallel port is a task many hardware enthusiasts have taken upon themselves.

In a unidirectional parallel communication system, there are two distinguished sites: the transmitting site and the receiving site. The transmitting site places its data on the data lines and informs the receiving site that data is available; the receiving site then reads the data lines and informs the transmitting site that it has taken the data. Note how the two sites synchronize their access to the data lines—the receiving site does not read the data lines until the transmitting site tells it to, and the transmitting site does not place a new value on the data lines until the receiving site removes the data

and tells the transmitting site that it has the data. In other words, this form of parallel communication between the printer and computer system relies on handshaking to coordinate the data transfer.

The PC's parallel port implements handshaking using three control signals in addition to the eight data lines. The transmitting site uses the *strobe* (or data strobe) line to tell the receiving site that data is available. The receiving site uses the *acknowledge* line to tell the transmitting site that it has taken the data. A third handshaking line, *busy*, tells the transmitting site that the receiving site is busy so it should not attempt to send data yet. The busy signal differs from the acknowledge signal in that acknowledge tells the system that the receiving site has accepted the data *and processed it*, whereas busy communicates only that the receiving site can't accept any new data yet—it does not imply that the last transmission has been processed (or even received).

In a typical data transmission session, the transmitting site:

1. Checks the busy line to see if the receiving site is busy. If the busy line is active, the transmitter waits in a loop until the busy line becomes inactive.
2. Places its data on the data lines.
3. Activates the strobe line.
4. Waits in a loop for the acknowledge line to become active.
5. Sets the strobe inactive.
6. Waits in a loop for the receiving site to set the acknowledge line inactive, indicating that it recognizes that the strobe line is now inactive.
7. Repeats steps 1 through 6 for each byte it must transmit.

Meanwhile, the receiving site:

1. Sets the busy line inactive when it is ready to accept data.
2. Waits in a loop until the strobe line becomes active.
3. Reads the data from the data lines.
4. Activates the acknowledge line.
5. Waits in a loop until the strobe line goes inactive.
6. Sets the busy line active (optional).
7. Sets the acknowledge line inactive.
8. Processes the data.
9. Sets the busy line inactive (optional).
10. Repeats steps 2 through 9 for each additional byte it receives.

By carefully following these steps, the receiving and transmitting sites coordinate their actions so that the transmitting site doesn't attempt to put several bytes on the data lines before the receiving site consumes them, and so the receiving site doesn't attempt to read data that the transmitting site has not sent.

15.1.3 Serial Ports

The RS-232 serial communication standard is probably the most popular serial communication scheme in the world. Although it suffers from many drawbacks (speed being the primary one), it is widely used, and there are thousands of devices you can connect to a PC using an RS-232 serial interface. Though many devices still use this standard, it is rapidly being eclipsed by USB (and today you can handle most RS-232 interfacing requirements by plugging a USB-to-RS232 cable into your PC).

The original PC system design supports concurrent use of up to four RS-232 compatible devices connected through the *COM1*;, *COM2*;, *COM3*;, and *COM4*; ports. To connect additional serial devices, you can buy interface cards that let you add 16 or more serial ports to the PC.

In the early days of the PC, DOS programmers had to directly access the 8250 serial communication controller (SCC) to implement RS-232 communications in their applications. A typical serial communications program would have a serial port ISR that read incoming data from the SCC and wrote outgoing data to the chip, as well as code to initialize the chip and to buffer incoming and outgoing data.

Fortunately, today's application programmers rarely program the SCC directly. Instead, OSes such as Windows and Linux provide sophisticated serial communications device drivers that application programmers can call. These drivers provide a consistent feature set that all applications can use, and this reduces the learning curve needed to provide serial communication functionality. Another advantage to the OS device driver approach is that it removes the dependency on the 8250 SCC. Applications that use an OS device driver will automatically work with different SCCs. In contrast, an application that programs the 8250 directly won't work on a system that uses a USB-to-RS232 converter cable. However, if the manufacturer of that converter cable provides an appropriate device driver for an OS, applications that do serial communications via that OS will automatically work with the USB/serial device.

An in-depth examination of RS-232 serial communications is beyond the scope of this book. For more information on this topic, consult your OS programmer's guide or pick up one of the many excellent texts devoted specifically to this subject.

15.2 Mice, Trackpads, and Other Pointing Devices

Along with disk drives, keyboards, and display devices, pointing devices are probably the most common peripherals you'll find on modern PCs. Pointing devices are among the least complex peripheral devices, providing a very simple data stream to the computer. They come in two categories: those that return the relative position of the pointer and those that return the absolute position of the pointing device. The *relative position* is the change in position since the last time the system read the device; the *absolute position* is some set of coordinate values within a fixed coordinate system. Mice, trackpads, and

trackballs return relative coordinates; touch screens, light pens, pressure-sensitive tablets, and joysticks return absolute coordinates.

Generally, it's easy to translate an absolute coordinate system to a relative one, but problematic to do the reverse. Converting a relative coordinate system to an absolute one requires a constant reference point that may become meaningless if, for example, someone lifts a mouse off the surface and sets it down elsewhere. Fortunately, most windowing systems work with relative coordinate values from pointing devices, so the limitations of pointing devices that return relative coordinates are not a problem.

Early mice were typically optomechanical devices that rotated two encoding wheels oriented along the x- and y-axes of the mouse body. Usually, both wheels were encoded to send 2-bit pulses whenever they moved a certain distance. One bit told the system that the wheel had moved a certain distance, and the other bit told the system which direction the wheel had moved.¹ By constantly tracking the 4 bits (2 bits for each axis) from the mouse, the computer system could determine the mouse's distance and direction traveled, and keep a very accurate record of the mouse's position in between application requests for that position.

One problem with having the CPU track each mouse movement is that, when moved quickly, mice can generate a constant and high-speed stream of data. If the system is busy with other computations, it might miss some of the incoming mouse data and therefore lose track of the mouse's position. Furthermore, the host's CPU time is better spent on application computations than tracking the mouse position.

As a result, mouse manufacturers decided early on to incorporate a simple microcontroller in the mouse package, to keep track of the physical mouse movements and respond to system requests for mouse coordinate updates, or at the very least generate interrupts on a periodic basis when the mouse position changes. Most modern mice connect to the system via the USB and respond with positional updates to system requests that occur about every 8 milliseconds.

Because of the wide acceptance of the mouse as a GUI pointing device, computer manufacturers have created many other devices that serve the same purpose but are more portable—mice aren't the most convenient pointing devices to attach to a laptop computer on the road, for example. Trackballs, strain gauges (the little “stick” between the *G* and *H* keys on many laptops), trackpads, trackpoints, and touch screens are all examples of devices that manufacturers have attached to laptop computers, tablets, and PDAs to create more portable pointing devices. Though these devices vary in their convenience to the end user, to the OS they can all look like a mouse. So, from a software perspective, there's little difference between them.

In modern OSes, the application rarely interfaces with a pointing device directly. Instead, the OS tracks the mouse position and updates cursors and other mouse effects in the system, then notifies the application when some sort of pointing device event (such as a button press) occurs.

1. This is a bit of a simplification, but we'll ignore that here.

In response to a query from an application, the OS returns the position of the system cursor and the state of the buttons on the pointing device.

15.3 Joysticks and Game Controllers

The analog game adapter created for the IBM PC allowed users to connect up to four resistive potentiometers and four digital switch connections to the PC. The design of the PC's game adapter was obviously influenced by the analog input capabilities of the Apple II computer, the most popular computer available at the time the PC was developed. IBM's analog input design, like Apple's, was designed to be dirt-cheap. Accuracy and performance were not a concern at all. In fact, you can purchase the electronic parts to build your own version of the game adapter, at retail, for less than \$3.

Due to the inherent inefficiencies of reading the original electronics of the IBM PC game controller, most modern game controllers contain the analog electronics that convert physical position into a digital value directly inside the controller, and then interface to the system via USB. Microsoft Windows and other modern OSes provide a special game-controller device-driver interface and APIs that allow applications to determine what facilities the game controller has, and also send the data to those applications in a standardized form. This allows game-controller manufacturers to provide many special features that were not possible with the original PC game-controller interface. Modern applications read game-controller data just as though they were reading data from a file or some other character-oriented device like a keyboard. This vastly simplifies the programming of such devices while improving overall system performance.

Some “old-school” game programmers feel that calling APIs is inherently inefficient and that great code always controls the hardware directly. This thinking is a bit outdated, for a few reasons. First, most modern OSes don't allow applications direct access to hardware even if the programmer wants it. Second, software that talks directly to the hardware won't work with as wide a variety of devices as software that lets the OS handle the hardware. Finally, most OS device drivers can probably be written more efficiently by the manufacturer's or OS developer's programming team than by an individual.

Because newer game controllers are no longer constrained by the design of the original IBM PC game-controller card, they provide a wide range of capabilities. Refer to the relevant game controller and OS documentation for information on how to program the API for a specific device.

15.4 Sound Cards

The original IBM PC included a built-in speaker that the CPU could program (using an onboard timer chip) to produce a single-frequency tone. Producing a wide range of sound effects was possible, but it required programming a single bit connected directly to the speaker. This process consumed nearly all the available CPU time. Within a couple of years of the PC's arrival, various manufacturers like Creative Labs created a special

interface board—a sound card—that provided higher-quality PC audio output and didn’t consume anywhere near that amount of CPU resources.

The first sound cards to appear for the PC didn’t follow any standards because none existed at the time. Creative Labs’ Sound Blaster card became the de facto standard because it had reasonable capabilities and sold in very high volumes. At the time, there was no such thing as a device driver for sound cards, so most applications were programming the registers directly on the sound card. Initially, so many applications were written for the Sound Blaster card that anyone wanting to use most audio applications also had to purchase it. Other sound card manufacturers quickly copied the Sound Blaster design, and all of them were subsequently stuck with it, because any new features they added wouldn’t be supported by the available audio software.

Sound card technology stagnated until Microsoft introduced multimedia support into Windows. The original audio cards were capable of mediocre music synthesis, suitable only for cheesy sound effects for video games. Some boards supported 8-bit telephone-quality audio sampling, but the audio was definitely not high fidelity. Once Windows provided a standardized, device-independent interface for audio, the sound card manufacturers began producing high-quality sound cards for the PC.

Immediately, “CD-quality” cards appeared that were capable of recording and playing back audio at 44.1 KHz and 16 bits. Higher-quality sound cards began adding *wavetable* synthesis hardware that produced realistic synthesis of musical instruments. Synthesizer manufacturers like Roland and Yamaha produced sound cards with the same electronics found in their high-end synthesizers. Today, professional recording studios use PC-based digital audio recording systems to record original music with 24-bit resolution at 96 (or even 192) KHz, arguably producing better results than all but the finest analog recording systems. Of course, such systems cost many thousands of dollars. They’re definitely not your typical sound card that retails for under \$100.

15.4.1 How Audio Interface Peripherals Produce Sound

Modern audio interface peripherals² generally produce sound in one of three different ways: analog (FM synthesis), digital wavetable synthesis, or digital playback. The first two schemes produce musical tones and are the basis for most computer-based synthesizers, while the third is used to play back audio that was digitally recorded.

The FM synthesis scheme is an older, lower-cost, music synthesis mechanism that creates musical tones by controlling various oscillators and other sound-producing circuits on the sound card. The sound produced by such devices is usually very low quality, reminiscent of early video games; there’s no mistaking it for an actual musical instrument. While some very low-end

2. The term *sound card* hardly applies anymore because many personal computers include the audio controller directly on the motherboard, and many high-end audio interface systems interface via USB or FireWire, or require multiple boxes and interface cards.

sound cards still use FM synthesis as their main sound-producing mechanism, few modern audio peripherals use it for anything other than producing intentionally “synthetic” sounds.

Modern sound cards that provide musical synthesis capabilities tend to use wavetable synthesis: the audio manufacturer typically records and digitizes several notes from an actual musical instrument, and then programs these digital recordings into read-only memory (ROM), which they assemble into the audio interface circuit. When an application requests that the audio interface play some note on a given musical instrument, the audio hardware plays back the recording from ROM, producing a very realistic sound.

However, wavetable synthesis is not simply a digital playback scheme. To record over 100 different instruments, each with a several octave range, would require a prohibitively expensive amount of ROM storage. Therefore, most manufacturers of such devices use software embedded on the audio interface card to raise or lower, by some integral number of octaves, a small number of digitized waveforms stored in ROM. This allows manufacturers to record and store only a single octave (12 notes) for each instrument. Some synthesizers use software to convert only a single recorded note into any other note, to reduce costs, but the more notes the manufacturer records, the better the quality of the resulting sound. Some of the higher-end audio boards record several octaves on complex musical instruments (like a piano) but only a few notes on some lesser-used, less complex sound-producing objects, like sound effects for gunshots, explosions, and crowd noise.

Finally, pure digital playback is used for two purposes: playing back arbitrary audio recordings and performing very high-end musical synthesis, known as *sampling*. A sampling synthesizer is, effectively, a RAM-based version of a wavetable synthesizer. Rather than storing digitized instruments in ROM, a sampling synthesizer stores them in system RAM. Whenever an application wants to play a given note from a musical instrument, the system fetches the recording for that note from system RAM and sends it to the audio circuitry for playback. Like wavetable synthesis methods, a sampling synthesizer can convert digitized notes up and down octaves, but because the system doesn’t have the cost-per-byte constraints associated with ROM, the audio manufacturer can usually record a wider range of samples from real-world musical instruments. Generally, sampling synthesizers provide a microphone input so you can create your own samples. This allows you, for example, to play a song by recording a barking dog and generating a couple octaves of “dog bark” notes on the synthesizer. Third parties often sell “sound fonts” containing high-quality samples of popular musical instruments.

The other use for pure digital playback is as a digital audio recorder. Almost every modern sound card has an audio input that will theoretically record “CD-quality” sound in stereo.³ This allows the user to record an

3. “CD quality” simply means that the board’s digitizing electronics are capable of capturing 44,100 16-bit samples every second. Usually the analog circuitry on the board doesn’t have sufficiently high quality to pass this audio quality through to the digitizing circuitry, so very few PC sound cards today are truly capable of “CD-quality” recording.

analog signal and play it back verbatim, like a tape recorder. With sufficient outboard gear, it's even possible to make your own musical recordings and burn your own music CDs, though to do so you'd want something a little bit fancier than a typical Sound Blaster card—something at least as advanced as the DigiDesign ProTools HDX or M-Audio system.

15.4.2 The Audio and MIDI File Formats

There are two standard mechanisms for playing back sound in a modern PC: audio file playback and MIDI file playback.

Audio files contain digitized samples of the sound to play back. While there are many different audio file formats (for example, WAV and AIF), the basic idea is the same—the file contains some header information that specifies the recording format (such as 16-bit 44.1 KHz, or 8-bit 22 KHz) and the number of samples, followed by the actual sound samples. Some of the simpler file formats allow you to dump the data directly to a typical sound card after proper initialization of the card; other formats may require a minor data translation before the sound card can process the data. In either case, the audio file format is essentially a hardware-independent version of the data you'd normally feed to a generic sound card.

One problem with sound files is that they can grow rather large. One minute of stereo CD-quality audio requires just less than 10MB of storage. A typical 3- to 4-minute song requires between 20MB and 45MB. Not only does such a file take up an inordinate amount of RAM, but it consumes a fair amount of storage in the software's distribution file as well. If you're playing back a unique audio sequence that you've recorded, you have no choice but to use this space to hold the sequence. However, if you're playing back an audio sequence that consists of a series of repeated sounds, you can use the same technique that sampling synthesizers use and store only one instance of each sound, and then use some sort of index value to indicate which sound you want to play. This can dramatically reduce the size of a music file.

This is exactly the idea behind the *Musical Instrument Digital Interface* (MIDI) file format. MIDI is a standard protocol for controlling music synthesis and other equipment. If you want to play back music that doesn't contain vocals or other nonmusical elements, MIDI can be very efficient.

Rather than holding audio samples, a MIDI file simply specifies the musical notes to play, when to play them, how long to play them, which instrument to play them on, and so on. Because it takes only a few bytes to specify all this information, a MIDI file can represent an entire song very compactly. High-quality MIDI files generally range from about 20KB to 100KB for a typical 3- to 4-minute song. Contrast this with the 20MB to 45MB for an audio file of the same length. Most sound cards today are capable of playing back *General MIDI* (GM) files using an on-board wavetable synthesizer or FM synthesis. Most synthesizer manufacturers use the GM standard to control their equipment, so its use is very widespread and GM files are easy to obtain.

One problem with MIDI is that the quality of the playback is dependent upon the quality of the end user's sound card. Some of the more expensive

audio boards do a very good job of playing back MIDI files, but some of the lower-cost boards—including, unfortunately, a large number of systems that have the audio interface built into the motherboard—produce cartoonish-sounding playback.

Therefore, you need to carefully consider using MIDI in your applications. On the one hand, MIDI offers the advantages of smaller files and faster processing. On the other hand, on some systems the audio quality will be quite low, making your application sound bad. You have to balance the pros and cons of these approaches for your particular application.

Because most modern sound cards are capable of playing back CD-quality recordings, you might wonder why the manufacturers don't collect a bunch of samples and simulate one of these sampling synthesizers. Well, they do. Roland, for example, provides the Virtual Sound Canvas program, which simulates its hardware Sound Canvas module in software. These virtual synthesizers produce very high-quality output, but consume a large percentage of the CPU's capability, thus leaving less power for your applications. If your applications don't need the full power of the CPU, these virtual synthesizers provide a very high-quality, low-cost solution.

If you know your target audience will have a synthesizer, another solution is to connect an outboard synthesizer module to your PC via a MIDI interface port and send the MIDI data to a synthesizer to play. This is an acceptable solution for a specialized application with a limited customer base, since few people outside of musicians would own a synthesizer.

15.4.3 Programming Audio Devices

One of the best aspects of audio in modern applications is that there's been a tremendous amount of standardization. File formats and audio hardware interfaces are very easy to use in modern applications. As with most other peripherals, few modern programs control audio hardware directly, because OSes like Windows and Linux provide device drivers that handle it for you. Producing sound in a typical Windows application requires little more than reading data from a file that contains the sound information and writing that data to another file used by the device driver, which interfaces with the actual audio hardware.

One other issue to consider when writing audio-based software is the availability of multimedia extensions in the CPU you're using. The Pentium and later 80x86 CPUs provide the MMX, SSE, and AVX instruction sets. Other CPU families provide comparable instruction set extensions (such as the AltiVec instructions on the PowerPC or NEON on ARM). Although the OS probably uses these extended instructions in the device driver, you can employ them in your own applications as well. Unfortunately, that usually involves assembly language programming, because few high-level languages provide efficient access to them. Therefore, if you're going to be doing high-performance multimedia programming, assembly language is something you'll probably want to learn. See *The Art of Assembly Language* for additional details on the Pentium's SSE/AVX instruction set.

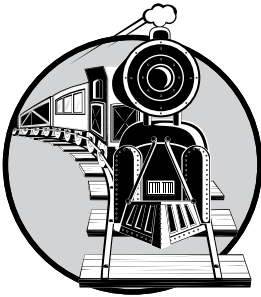
15.5 For More Information

Axelson, Jan. *Parallel Port Complete: Programming, Interfacing, & Using the PC's Parallel Printer Port*. Madison, WI: Lakeview Publishing, 2000.

———. *Serial Port Complete: Programming and Circuits for RS-232 and RS-485 Links and Networks*. Madison, WI: Lakeview Publishing, 2000.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

AFTERWORD: THINKING LOW-LEVEL, WRITING HIGH-LEVEL



The goal of this book was to get you thinking at the machine level. One way to force yourself to write code at this level is to write your applications in assembly language. When you write code statement by statement in assembly language, you get a pretty good idea of the cost associated with each one.

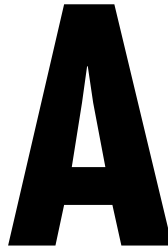
Unfortunately, using assembly language isn't a realistic solution for most applications. The disadvantages of assembly language have been well publicized (and exaggerated) over the past several decades, and as a result many people have decided assembly isn't an option for them.

Unlike writing code in assembly language, writing code in a high-level language doesn't force you to think at a high level of abstraction. There's nothing preventing you from thinking in low-level terms while writing high-level code. This book has equipped you with the background knowledge you need to do just that. By learning how the computer represents data, you've learned how HLL data types translate to the machine level.

By learning how the CPU executes machine instructions, you've learned the costs of various operations in your HLL applications. And by learning about memory performance, you've learned how to organize your HLL variables and other data to maximize cache and memory access. There's only one piece missing from this puzzle: "Exactly *how* does a particular compiler map HLL statements to the machine level?" That topic is sufficiently large that it deserves an entire book on its own. And that's the purpose of the second volume in the *Write Great Code* series: *Thinking Low-Level, Writing High-Level*.

WGC2 will pick up right where this book leaves off. It will teach you how each statement in a typical HLL maps to machine code, how you can choose between two or more high-level sequences to produce the best possible machine code, and how to analyze that machine code to determine its quality and that of the high-level code that produced it. And while doing all of this, it will give you a greater appreciation for how compilers work and encourage you to help them do their job better.

Congratulations on your progress thus far toward writing great code. See you in Volume 2.



ASCII CHARACTER SET

Binary	Hexadecimal	Decimal	Character
0000_0000	00	0	NULL
0000_0001	01	1	CTRL A
0000_0010	02	2	CTRL B
0000_0011	03	3	CTRL C
0000_0100	04	4	CTRL D
0000_0101	05	5	CTRL E
0000_0110	06	6	CTRL F
0000_0111	07	7	Bell
0000_1000	08	8	Backspace
0000_1001	09	9	TAB
0000_1010	0A	10	Line feed
0000_1011	0B	11	CTRL K
0000_1100	0C	12	Form feed
0000_1101	0D	13	RETURN
0000_1110	0E	14	CTRL N

Binary	Hexadecimal	Decimal	Character
0000_1111	0F	15	CTRL O
0001_0000	10	16	CTRL P
0001_0001	11	17	CTRL Q
0001_0010	12	18	CTRL R
0001_0011	13	19	CTRL S
0001_0100	14	20	CTRL T
0001_0101	15	21	CTRL U
0001_0110	16	22	CTRL V
0001_0111	17	23	CTRL W
0001_1000	18	24	CTRL X
0001_1001	19	25	CTRL Y
0001_1010	1A	26	CTRL Z
0001_1011	1B	27	CTRL [
0001_1100	1C	28	CTRL \
0001_1101	1D	29	ESC
0001_1110	1E	30	CTRL ^
0001_1111	1F	31	CTRL _
0010_0000	20	32	Space
0010_0001	21	33	!
0010_0010	22	34	"
0010_0011	23	35	#
0010_0100	24	36	\$
0010_0101	25	37	%
0010_0110	26	38	&
0010_0111	27	39	'
0010_1000	28	40	(
0010_1001	29	41	)
0010_1010	2A	42	*
0010_1011	2B	43	+
0010_1100	2C	44	,
0010_1101	2D	45	-
0010_1110	2E	46	.
0010_1111	2F	47	/
0011_0000	30	48	0
0011_0001	31	49	1
0011_0010	32	50	2
0011_0011	33	51	3
0011_0100	34	52	4

Binary	Hexadecimal	Decimal	Character
0011_0101	35	53	5
0011_0110	36	54	6
0011_0111	37	55	7
0011_1000	38	56	8
0011_1001	39	57	9
0011_1010	3A	58	:
0011_1011	3B	59	;
0011_1100	3C	60	<
0011_1101	3D	61	=
0011_1110	3E	62	>
0011_1111	3F	63	?
0100_0000	40	64	@
0100_0001	41	65	A
0100_0010	42	66	B
0100_0011	43	67	C
0100_0100	44	68	D
0100_0101	45	69	E
0100_0110	46	70	F
0100_0111	47	71	G
0100_1000	48	72	H
0100_1001	49	73	I
0100_1010	4A	74	J
0100_1011	4B	75	K
0100_1100	4C	76	L
0100_1101	4D	77	M
0100_1110	4E	78	N
0100_1111	4F	79	O
0101_0000	50	80	P
0101_0001	51	81	Q
0101_0010	52	82	R
0101_0011	53	83	S
0101_0100	54	84	T
0101_0101	55	85	U
0101_0110	56	86	V
0101_0111	57	87	W
0101_1000	58	88	X
0101_1001	59	89	Y
0101_1010	5A	90	Z

Binary	Hexadecimal	Decimal	Character
0101_1011	5B	91	[
0101_1100	5C	92	\
0101_1101	5D	93	]
0101_1110	5E	94	^
0101_1111	5F	95	_
0110_0000	60	96	`
0110_0001	61	97	a
0110_0010	62	98	b
0110_0011	63	99	c
0110_0100	64	100	d
0110_0101	65	101	e
0110_0110	66	102	f
0110_0111	67	103	g
0110_1000	68	104	h
0110_1001	69	105	i
0110_1010	6A	106	j
0110_1011	6B	107	k
0110_1100	6C	108	l
0110_1101	6D	109	m
0110_1110	6E	110	n
0110_1111	6F	111	o
0111_0000	70	112	p
0111_0001	71	113	q
0111_0010	72	114	r
0111_0011	73	115	s
0111_0100	74	116	t
0111_0101	75	117	u
0111_0110	76	118	v
0111_0111	77	119	w
0111_1000	78	120	x
0111_1001	79	121	y
0111_1010	7A	122	z
0111_1011	7B	123	{
0111_1100	7C	124	
0111_1101	7D	125	}
0111_1110	7E	126	~
0111_1111	7F	127	

GLOSSARY

A

ABI Application binary interface

Abstract base class A class that has at least one abstract method (member function).

Abstract method A method in a class that does not have an implementation. Derived classes are responsible for implementing the abstract method.

Accuracy The correctness of a computation.

Activation record A block of memory holding parameters, local variables, and other memory entities associated with a procedure/function call.

Address bus The portion of the system bus on which memory addresses appear (to access memory and I/O devices).

Address space The range of memory locations available to a single application.

Addressing modes A mechanism for selecting an effective address in memory by combining register values, constants, and other components.

Aggregate data type A data type containing a collection of data values.

AGP Accelerated graphics port

ALU Arithmetic logical unit

Anonymous variables Variables without a name bound to them. For example, a data structure that a program allocates on the heap and refers to via a pointer (rather than by name) is an anonymous variable.

Arabic numerals The common 10 digits (0–9) used by most Western countries to represent base-10 values.

Architecture (computer) See *Computer architecture*.

ARM Acorn RISC Machine; a popular CPU architecture (used in most smartphones, for example).

Array base address Memory address of the first element of an array.

Associated values In Swift, an associated value is an auxiliary value associated with an enum constant in an enumerated data type. Swift uses associated values to provide the same functionality as discriminate unions or variant record data types.

Associativity A binary operator $\circ$ is said to be associative if $(A \circ B) \circ C = A \circ (B \circ C)$ for all Boolean values A , B , and C .

Asynchronous I/O I/O operations that take place independent of the CPU's activities. That is, the CPU starts the I/O operation and then performs other activities without waiting for the I/O operation to complete.

ATA Advanced Technology Attachment; an older disk-drive-interface command set. SATA is the modern replacement.

ATAPI ATA with Packet Interface

Average rotational latency The average time required for a desired disk sector to appear under a disk head.

B

Base class An ancestor class of derived classes.

Basic Multilingual Plane The first group of 65,536 Unicode code points (U+0000 to U+CFFF and U+E000 to U+FFFF).

BCD Binary-coded decimal

Best-fit A memory allocation scheme whereby the system scans all the blocks on the free list to find the smallest one that can satisfy the allocation request.

Big-endian A byte organization (usually in memory) where the HO byte of a multi-byte structure appears first in the string of bytes (for example, the HO byte appears at the lowest address in memory and the LO byte appears at the highest address).

Binary-coded decimal A binary representation of base-10 numbers that uses nibbles (4 bits) to represent a single decimal digit. Binary values 1010 through 1111 are illegal BCD values.

Binary numbering system A numeric system based on powers of 2 (and only having the digits 0 and 1).

Binding Associating some attribute with an object (such as associating a value with a variable).

Bit A single binary digit, representing the value 0 or 1.

Bit strings An ordered sequence of one or more binary digits (bits).

Bitwise Operations on two bit strings that proceed on a bit-by-bit basis; that is, they operate on 2 bits at a time, each of which occupies the same position in its respective bit string.

BIU Bus interface unit

BMP See *Basic Multilingual Plane*.

Boolean expressions Arithmetic-like expressions that evaluate to either true or false.

Boolean logic A mathematical system based on two values (for example, 0 and 1, or true and false).

BSS Block started by a symbol; an area of memory bound to an identifier.

Byte A bit string containing exactly 8 bits.

Byte-addressable memory Memory from which the CPU can access individual bytes (as opposed to other memory, often on RISC processors, that can be accessed only 32 or 64 bits at a time).

C

Cache hit When a CPU accesses a memory location that is present in the memory cache.

Cache line A group of memory locations managed as a set by the CPU or cache manager. Typically, a CPU writes or reads an entire cache line to/from memory at one time.

Cache miss A memory access to a location that is not currently held in cache memory.

Canonical equivalence Two different sequences (such as strings) are canonically equivalent if they produce the same character on an output device. If two strings are canonically equivalent, comparing them for equality should produce true even if they have different bytes in their sequences.

Casting The process of converting a value from one type to another.

Central processing unit The core component in a computer system where arithmetic, logical, control, instruction fetch and decode, and other operations take place.

Character string A sequence of characters.

CISC Complex instruction set computer

Clock frequency The frequency of the signal (typically a square wave) input to a CPU that synchronizes and controls its internal operations. The speed of the CPU is often directly proportional to the clock frequency.

Closure A mathematical system is closed with respect to a particular operator if every pair of values in that system supplied to that operator produces a value in the system.

Code pages Different character sets sharing the same numeric representation (for example, multiple EBCDIC character sets lie in different code pages).

Code plane A set of up to 65,536 different Unicode characters.

Code point A numeric value (in the range 0–65,535) representing a Unicode character (scalar) or a surrogate code point (Unicode character set expansion).

Column-major ordering Organizing arrays in memory with column elements appearing in consecutive memory locations.

Commutative A two-operand operator is commutative if you can swap the left and right operands and the operator produces the same result.

Composite data types Data types that contain a collection of other data objects. Examples include arrays, structs/records, classes, tuples, and unions.

Computer architecture The set of rules and methods that define the functionality and organization of a computer system.

Control bus The portion of the system bus that contains various control lines, such as read/write control, byte enable lines, clock signals, and hold lines.

Control characters Special characters that perform terminal/device manipulation rather than displaying a symbol on an output device. Examples include backspace, carriage return, and line feed.

Control hazard An attempt by a CPU to continue executing instructions in a pipeline after a control transfer occurs (invalidating the sequential instructions in the pipeline).

Copy on write A mechanism whereby data is shared between multiple variables until one variable writes to the common data, at which point the system makes a copy of the data prior to modifying it (for the variable writing to the common data).

Core See *CPU core*.

CPU Central processing unit

CPU clock A signal that controls the rate of activities within a CPU (also known as the *system clock*). See also *Clock frequency*.

CPU core A full CPU on a single piece of silicon. Often, multiple CPU cores appear on the same piece of silicon, allowing multiple threads of concurrent execution.

CU Control unit

D

Data bus The portion of the system bus where various components exchange data with one another.

Data hazard An attempt by a CPU to use a piece of data before a currently executing instruction is done using that data.

DBCS Double-byte character set; a character set scheme that uses 1 or 2 bytes to represent a large number of characters (typically fewer than 500 characters total).

Decimal numbering system A numeric representation system based on exponents of 10.

Delphi A popular Object Pascal compiler and development system.

Denormalized values Floating-point numbers whose exponent contains 0 and where the binary point isn't between the HO mantissa bit position and bit position HO - 1.

Descriptor A record (structure) that maintains information about a data structure somewhere else in memory. For example, a string descriptor might contain the length of the string along with a pointer to the characters in that string.

Direct addressing Accessing a memory location using an address encoded as part of the machine instruction.

Dirty bit A flag in a cache line specifying that data has been written to the cache line but not all the way through to main memory.

Discriminant union Also known as a *discriminated union*. A collection of data values in a structure whose use is mutually exclusive. That is, for the lifetime of the object, the software will reference only a single field from the structure. Often, compilers will allocate all the fields of a union at the same memory address to conserve memory (because only one should ever be used at a time).

Distributive Two binary operators $\circ$ and $\%$ are distributive if $A \circ (B \% C) = (A \circ B) \% (A \circ C)$ for all Boolean values A , B , and C .

DMA Direct memory access

DOS Disk operating system

Double word A bit string that is the size of 2 words (typically 32 bits, but it could be larger if the CPU's native word size is greater than 16 bits).

DRAM Dynamic random-access memory; the most common form of memory in modern computer systems.

DSM Distributed shared memory

Dword See *Double word*.

Dynamic objects Objects that have some attribute bound to them while a program is running.

Dynamic range The difference between the smallest and largest numbers in a given numeric representation.

E

EBCDIC Extended Binary Coded Decimal Interchange Code

Effective address A memory address resulting from the computation of all addressing mode components in an instruction.

Endianness The organization of bytes in a multibyte data object in memory. Big-endian organization stores the HO byte of a data structure at the lowest (byte) address in memory, whereas the LO byte appears at the highest address. Little-endian organization stores the LO byte at the lowest address in memory and the HO byte at the highest address.

Excess-127 format A binary representation of floating-point exponents using 8 bits, with values 0 through 127 representing negative exponents and 128 through 255 representing positive exponents (and 0).

Excess-1,023 exponent A binary representation of floating-point exponents using 11 bits, with values 0 through 1,023 representing negative exponents and 1,024 through 2,047 representing positive exponents (and 0).

Excess-16,383 exponent A binary representation of floating-point exponents using 15 bits, with values 0 through 16,383 representing negative exponents and 16,384 through 32,767 representing positive exponents (and 0).

Exponent A mantissa is multiplied by the numeric base (usually binary) raised to the exponent power (for example, for *m.mmmmm+xx* the exponent is the *xx* portion).

F

Falling edge The component of a clock signal where the signal changes from high to low.

False precision Garbage bits generated by certain computations resulting in imprecise results.

FAT File allocation table

Field A data memory of a record, structure, class, or other aggregate/composite data type.

FIFO First in, first out

First-fit A memory allocation scheme in which the memory manager allocates first block (in the free-block list) that satisfies the allocation request.

Fixed-point representation A numeric representation using a fixed set of digits with a radix point at an assumed position in the string of digits. For example, a six-digit decimal fixed-point representation (with the decimal point fixed between the third and fourth digits) could represent values between 000.000 and 999.999. Most commonly, fixed-point values in computers are binary fixed-point values, with the binary point fixed at a certain location in the bit string.

Floating-point representation A numeric representation for real numbers that contains two components: a mantissa and an exponent.

FPU Floating-point unit

Fragmentation In a memory allocation scheme, fragmentation occurs when larger blocks are broken up into small blocks that aren't sufficient to handle common allocation requests.

G

Garbage collection Automatic reclamation of dynamically allocated memory by a system.

Glyph A set of strokes that draw a character on an output device.

GM General MIDI

Grapheme clusters A sequence of Unicode code points that produce a single item most people would recognize as a stand-alone character on an output device.

H

Harvard architecture A CPU architecture that uses separate memory spaces for code and data.

Hazards Attempts by a CPU to simultaneously use a single resource by multiple instructions.

Heap A region in memory reserved for dynamic storage allocation.

Hexadecimal numbering system A numbering system based on powers of 16.

HLA High-Level Assembly (language)

HO High-order (most significant)

Hyperthreading A scheme (typically in 80x86 processors) whereby multiple threads of execution occur in parallel by using functional units on the CPU that are currently idle.

Hz Hertz (also known as *cycles per second*)—the unit of the system clock frequency.

I

I/O Input/output

IDE Integrated drive electronics; an older disk drive interface (SATA is the modern equivalent).

Identity A Boolean value I is said to be the identity element with respect to some binary operator $\circ$ if $A \circ I = A$ for all Boolean values A .

Immediate operand A constant operand for a machine instruction.

Indexed addressing Computing the effective address by adding an index (numeric value) held in a machine register or memory location to some base address (which could also be in a register, in a memory location, or encoded as part of the machine instruction).

Indexed addressing mode A memory address computed by adding some value (typically held in a register) to a base address.

Indirect addressing Referencing a memory location by an address held in a register or another memory location (that is, using a pointer to the memory location).

Indirect addressing mode Accessing an address in memory where the address is held in some register or memory location (rather than encoded directly in the machine instruction).

Inheritance The process of one class inheriting attributes and behaviors from another class (also known as *subclassing*).

Instruction cache A high-speed cache memory used to hold machine instructions.

Instruction set architecture The design of the machine instruction set for a CPU.

Inverse A Boolean value I is said to be the inverse element with respect to some binary operator $\circ$ if $A \circ I = B$ and $B \neq A$ (that is, B is the opposite value of A in a Boolean system) for all Boolean values A and B .

ISA Instruction set architecture; also the Industry Standard Architecture bus (the name of the original IBM PC bus).

ISR Interrupt service routine

K

Kylix A Linux-based version of Delphi (Object Pascal).

L

L1 cache Level 1 caching system.

L2 cache Level 2 caching system.

L3 cache Level 3 caching system.

Latency The time between a request for some resource (such as data in cache memory) and the actual fulfillment of that request.

Least significant bit The bit in a bit string representing the smallest value (smallest exponent of 2). Typically the rightmost bit in a bit string.

Length-prefixed string A string that begins with a count of the number of characters in the string.

Lifetime The period of time between the binding of some attribute to the point when the bond is broken. For example, the lifetime of a memory variable is usually from the point you allocate memory for the variable to the point you deallocate that storage.

Little-endian A byte organization (usually in memory) where the LO byte of a multi-byte structure appears first in the string of bytes (for example, the LO byte appears at the lowest address in memory and the HO byte appears at the highest address).

LO Low-order (least significant)

Long word A bit string whose size is 128 bits.

LRU Least recently used

LSB Least significant bit

M

Machine code A numeric encoding of machine instructions in memory.

Machine instructions Commands that a CPU executes natively.

Macroinstruction Native CPU machine instructions (emulated by executing several microcode instructions on microcoded CPUs).

Mantissa The significant digits in a real number that do not include the exponent portion (for example, for *m.mmmme + xx* the mantissa is the *m.mmmm* portion).

Mask in Force bits to 1 in a bit string.

Mask out Force bits to 0 in a bit string.

MASM Microsoft Macro Assembler

Memory access time The amount of time a CPU takes to fetch (or write) a memory element. Typically specified in system clock period units (that is, nanoseconds or picoseconds), though operating frequency (for example, GHz) is also common (clock period is the reciprocal of the clock frequency).

Memory addressing modes A mechanism for computing memory addresses on a CPU.

Memory controllers Specialized components (typically on modern CPUs) that interface directly to DRAM devices providing appropriate address/data multiplexing, refresh control, and other memory-related functions.

Memory leak Making dynamically allocated memory unavailable for reuse. Occurs when code stops using an allocated block of memory without explicitly freeing it.

Memory-mapped files Storing files in the address space of a process and accessing the file data using virtual memory operations.

MHz Megahertz, or one million cycles per second

Microcode Internal low-level code that a CPU executes in order to execute native machine instructions.

Microengine The component of the CPU that executes microcode.

Microinstructions Low-level instructions in microcode.

Microsecond One millionth of a second.

MIDI Musical Instrument Digital Interface

Millisecond One thousandth of a second.

MIMD Multiple instructions, multiple data

MMC Multimedia Commands (SCSI)

MMU Memory management unit

Modulo-*n* counters Variables that increment from 0 to (*n* – 1) and then reset to 0.

Most significant bit The bit with the greatest value in a bit string (often the leftmost bit in a bit string).

MSB Most significant bit

MSC Management Server Commands (SCSI)

Multiple inheritance The ability for a class to inherit attributes (data fields) and behaviors (methods/functions) from multiple parent classes.

Multiprocessing Executing multiple threads of execution on multiple CPUs (or CPU cores).

N

NaN Not-a-number; a special floating-point representation for illegal values.

Nanosecond One billionth of a second.

Nibble A bit string containing exactly 4 bits.

Normalized floating-point values

Floating-point numbers that have their exponents adjusted so that the binary point is between the HO and HO – 1 bit positions in the mantissa.

Nsec See *Nanosecond*.

NUMA Non-uniform memory access

Number An intangible concept that represents a quantity.

Numbering system A set of symbols and conventions for representing numeric values.

Numeric representation Printable symbols human beings use to represent numbers.

O

Octal numbering system A numeric representation based on powers of 8.

Offline storage Information kept on media that is not connected to the computer system that uses it; examples include magnetic tapes and optical disks.

One's complement format A signed numeric representation that uses a single bit as a sign bit. Note that one's complement has two representations of 0 (with the sign bit containing 0 or 1).

Opcode Operation codes; numerical encodings for machine instructions.

Operation codes See *Opcodes*.

Operator precedence See *Precedence*.

OS Operating system

OSD Object-based Storage Device commands (SCSI)

Out-of-order execution Certain CPUs delay completing the execution of certain instructions until after later instructions have begun executing; however, the CPUs (usually) attempt to ensure that the results produced with out-of-order execution are the same as if they'd been produced with linear execution.

Overflow Incorrect calculations that produce a value that is too large to fit in the destination bit string.

P

Parallel processing Running multiple threads (programs) concurrently on multiple CPUs (or CPU cores).

Parameterized type Specifying a type as a parameter (argument) to a class definition or function.

PATA Parallel ATA (same as IDE/ATA)

PCI Peripheral component interconnect

Pipeline Stages in a CPU's hardware that execute phases of a machine instruction.

Pointer A variable whose value refers to a different data value. Typically pointers contain the memory address of the object they reference.

Polling Software testing to see if a resource or operation is available.

Polymorphism The feature of an object-oriented programming language whereby an object reference to some base class could actually refer to a derived class object. A polymorphic type is one whose operations can apply to other types.

Positional notation system A system (typically numeric) where different positions in a string of characters stand for different entities. For example, the decimal numbering system is positional, with each digit to the left representing a greater power of 10.

Postulate An initial assumption in a mathematical system.

Powerset A set of objects using a bit string representation with 1 bit (present/not present) for each possible member of the set.

Pragma A special programming language feature that provides information to the compiler (rather than generating machine code, as is the case for normal statements).

Precedence A property that controls the order of evaluation when multiple operators appear within an expression.

Precision The number of digits or bits maintained in a computation.

Prefetch queue A special first-in, first-out memory inside a CPU that holds machine instructions that the CPU is about to execute.

Prefix opcode byte A prefix byte in a machine instruction that redefines the following opcodes.

Q

QNaN Quiet not-a-number

Quad word A bit string that is 4 words concatenated together; this is usually 64 bits.

R

Radix Base of a numbering system. For example, Radix-10 is the decimal (base-10) numbering system.

Radix point A period that separates whole numbers from fractional values in a numeric representation. Usually, the base (such as decimal or hexadecimal) name is used rather than “radix” (for example, *decimal point* or *hexadecimal point*).

RAID Redundant array of inexpensive disks

RAM Random access memory

Rational representation A numeric representation for fractional numbers that uses two integers to represent the numerator and denominator of a fractional value.

RBC Reduced Block Commands (SCSI)

Record In a language like Pascal, a composite data type that allows you to combine different data objects into a single type.

Reference counter A data structure used to count how many pieces of code are using (referencing) a block of memory so that the memory can be reclaimed (garbage-collected) once the code is done using it.

Register renaming An architectural feature of the CPU that allows it to process some operations faster by using shadow registers in place of its main registers when the main registers are unavailable.

RISC Reduced instruction set computer; a computer architecture based around reducing the work each machine instruction performs.

Rising edge The component of a clock signal where the signal changes from low to high.

Row-major ordering Organizing elements of an array with elements in rows appearing in consecutive memory locations.

S

SAS Serial-Attached SCSI

Saturation The process of storing a maximum (or minimum) value into a numeric variable if the range of the original value does not fit in the variable.

SBC SCSI Block Commands, or single-board computer

Scaled-index addressing Like an indexed address mode, but the scaled-index addressing mode multiplies the index value by some constant (usually 2, 4, 8, or 16) prior to adding the index to the base address.

Scaled numeric format A numeric representation that multiplies (or divides) all values by a fixed constant. For example, a “times 1,000” scaled format would multiply all numbers by 1,000, allowing an integer value to represent numeric quantities from $x.000$ to $x.999$ (where x is some arbitrary integer value).

SCC SCSI Controller Commands, or serial communications chip

Scope The portion of a program where an identifier’s name is bound to an object.

SCSI Small Systems Computer Interface; an older interface to hard drives and other peripherals. SAS (Serial-Attached SCSI) is the modern implementation of SCSI.

SES SCSI Enclosure Services commands

SGC SCSI Graphics Commands

Sign contraction The process of reducing the number of bits used by a two’s complement signed integer (sign contraction isn’t always possible if the value won’t fit into the reduced number of bits).

Sign extension The process of expanding the size (bits) of a two’s complement signed integer.

SIMD Single instruction, multiple data

SISD Single instruction, single data

SNaN Signaling not-a-number

Spatial locality The idea that if a system accesses a given memory location, it will likely access an adjacent memory location in the near future (such as accessing successive machine instructions in memory).

SPC SCSI Primary Commands

SPI SCSI Parallel Interface

SSC SCSI Stream Commands

SSD Solid-state drive; a semiconductor-based mass storage replacement for hard drives.

Static (binding) Objects exhibit static binding when an attribute is bound to the object the whole time the program is running.

Superscalar CPU A CPU that is capable of executing more than one instruction simultaneously.

Surrogate code points Special Unicode values that expand the character set beyond 65,536 characters (expansion beyond 16 bits).

Synchronous I/O With synchronous I/O, a process starts an I/O operation and then waits for its completion before continuing execution.

System bus A set of signal lines that connect various components of a computer system (such as the CPU, memory, and I/O devices).

T

Tbyte A bit string whose size is 80 bits.

Temporal locality The idea that if a system accesses a given memory location, it likely will access this same location in the near future.

Thrashing Repeated accesses to memory objects that are either not in the cache or not in physical memory (forcing a reload of the cache from main memory or a reload of memory from a secondary storage device, such as a hard drive), resulting in reduced system performance.

Tuple A list of associated data values. In Swift, a tuple is roughly equivalent to a list of values.

Two's complement representation A special binary format representing signed and unsigned integers.

U

Underflow Incorrect results produced from a calculation that are too small to fit in the destination bit string.

Unicode A universal standardized character set that supports most known characters.

Unicode normalization Adjusting canonically equivalent Unicode strings so that they have the same (minimal) code points, organized in the same order.

Union See *Discriminant union*.

USB Universal Serial Bus

μsec See *Microsecond*.

UTF Universal Transformation Format; an encoding scheme for Unicode (UTF-8, UTF-16, and UTF-32 are the three standard Unicode encoding schemes).

V

Virtual memory Utilizing secondary storage (hard drives or SSDs) to hold infrequently accessed data so main memory is available for frequently used data.

Virtual method table A table of pointers, each of which contains the address of a virtual method associated with a class. Objects use virtual method tables to provide linkages to methods associated with the underlying class to supply polymorphism.

VLIW Very-long instruction word; a high-performance computer architecture.

VMT See *Virtual method table*.

Von Neumann architecture A computer architecture for a stored-program system where the data and program codes sit in the same memory space and the computer fetches both on the same address and data bus.

W

Wait state A clock cycle during which the CPU suspends activities while waiting to synchronize with external hardware (such as slow memory).

WGC *Write Great Code*

WGC1 *Write Great Code, Volume 1: Understanding the Machine*

WGC2 *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level*

WGC3 *Write Great Code, Volume 3: Engineering Software*

WGC4 *Write Great Code, Volume 4: Designing Great Code*

WGC5 *Write Great Code, Volume 5: Great Coding*

WGC6 *Write Great Code, Volume 6: Testing, Debugging, and Quality Assurance*

Word A bit string of some CPU-native length. Typically 16 bits on modern CPUs, but it could be 32 or even 64 bits.

Z

Zero extension The process of expanding an unsigned binary bit string to a larger-sized bit string.

Zero-terminated strings A character string that contains a 0 byte as the last element of the string.

INDEX

Numbers

- 7-bit strings, 111
 - advantages, 112
 - assembly language macro
 - implementation, 112
- 16-bit bus data access, 138
- 8042 microcontroller chip (keyboard controller), 415

A

- absolute position pointing devices, 417
- abstract base classes, 202
- abstract member functions in C++, 206
- abstract methods, 202
- Accelerated Graphics Port (AGP)
 - bus, 360
- accessing data with a 16-bit bus, 138
- accessing double words in memory, 140
- accessing elements of an array, 171, 175
- accessing words
 - in byte-addressable memory, 136
 - at odd addresses, 139
- acknowledge line (parallel port), 416
- activation records, 342
- active-low logic, 242
- adders, 240
- adding integer values to a pointer, 164
- add instruction, 257
 - encoding on the x86, 310
 - encoding on the Y86, 296
- addressable memory, 133
- address bus, 133
- address spaces, 135
- Advanced Technology Attachment (ATA), 373
- advantages of 7-bit strings, 112
- AGP (Accelerated Graphics Port)
 - bus, 360
- algebraic manipulation of Boolean expressions, 223
- aliases, 191
- aligned data access, 164
- allocating objects in contiguous memory, 163
- AND operation, 42, 218
- anonymous variables, 161, 342
- application binary interface (ABI), 185
- Arabic numerals, 11
- architecture, 131
- arithmetic and logical instructions (Y86), 292
- arithmetic shift right operation, 49
- arithmetic units, 263
- ARM CPU (memory access), 141
- arrays, 166
 - alignment in memory, 170
 - of arrays, 179
 - in C#, 168
 - declarations, 167
 - implementation in Swift, 179
 - index bounds checking, 169
 - initialization in Swift, 168
 - Pascal, 169
 - representation in memory, 170
- Art of Assembly Language*, 7, 59, 93, 157, 215, 317, 424
- ASCII character set, 96–98
- assembly language macro to declare 7-bit strings, 112
- assigning instruction opcodes, 290
- associativity, 219
- asynchronous I/O (filesystems), 409
- ATA (Advanced Technology Attachment), 373
- ATAPI (ATA with Packet Interface), 373
- audio device programming, 423
- audio sampling, 421
- average rotational latency (of a disk drive), 386
- average seek time (on a disk), 384

B

- backspace character, 97
- base addresses
 - of an allocated memory region, 162
 - of an array, 166
 - of a record, 185
- base classes, 195
- base (numbering system radix), 12
- BCD, 29
- BDXL, 391
- best-fit memory allocation, 343
- biased (excess) exponents, 68
- bidirectional ports, 352
- big-endian data organization, 142
 - issues when using unions, 192
- binary arithmetic
 - addition, 38
 - division, 41
 - multiplication, 40
 - subtraction, 39
- binary-coded decimal, 29, 99
- binary conversion
 - to decimal, 13
 - to hexadecimal, 16
- binary data types, 20
- binary I/O (files), 409
- binary numbering system, 13
- binary operator, 218
- binary representations for NaN, 73
- bit density (on a disk), 384
- bit fields, 51, 54
- bit numbers, 21
- bit strings, 20, 44
- bitwise operations, 44
- BIU (bus interface unit), 265
- blocks (on a disk), 383
- blowing revs, 386
- Blu-ray, 391
- BMP (Unicode Basic Multilingual Plane), 102
- Boolean algebra, 218
 - theorems of, 219
- Boolean expressions, 220, 223
- Boolean function numbers, 222
- Boolean functions, 220, 221
- Boolean literals, 223
- Boolean logic, 217
- Boolean map simplification, 229
- Boolean operators, 218
 - precedence, 220
- Boolean postulates, 218
- Boolean terms, 223, 224

- bounds checking of array indexes, 169
- buffering peripheral device data, 360
- bulk transmissions (USB), 378
- burst mode on the PCI bus, 359
- bus contention, 270
- bus interface unit (BIU), 265
- busy line (parallel port), 416
- byte-addressable memory, 135, 162
- byte-addressable memory array, 138
- byte enable lines, 135, 140
- bytes, 20

C

- C/C++ records (structs), 182
- C/C++ unions, 187
- C# strings, 115
- caches
 - coherency, 280
 - disk, 387
 - hits, 152, 153
 - line replacement policies, 329
 - lines, 152, 326
 - memory, 151
 - misses, 152
 - three-level (L3), 154
 - two-level (L2), 153
 - write policies, 330
- canonical equivalence (Unicode), 105
- canonical forms of Boolean
 - expressions, 224
- carriage return character, 97
- case-variant records in Pascal/
Delphi, 187
- CD-Recordable (CD-R), 391
- CD-Rewritable (CD-RW), 391
- CD-ROM, 390
- ceil() function, 72
- central processing unit (CPU), 131, 154
- character names in Unicode, 103
- character sets, 96–101, 119, 120
 - ASCII, 96–99
 - double-byte, 100
 - EBCDIC, 99–100
 - Unicode, 101
- character strings, 110
- choosing instructions for a CPU, 289
- class constructors, 198
- classes, 192
 - in C++, 205
 - in Java, 208
 - in Swift, 209
- client drivers (USB), 380

- clipping (saturation), 28
- clock cycle, 148
- clocked logic, 245
- clocks per instruction (CPI), 275
- closure (of an operator), 218
- coarse-grained parallelism, 280
- code planes in Unicode, 102
- code points in Unicode, 101
- column-major ordering, 173, 176
- combining characters in Unicode, 108, 109
 - acute accent character, 104
- COM ports, 417
- commutivity, 218
- comparing bits, 46
- comparing dates, 53
- comparing floating-point numbers, 65
- comparing pointers, 162, 166
- complex instruction set computer (CISC) instructions, 255, 301
- composite data types, 166
- computer architecture, 131
- conditional jumps, 255
 - on the Y86, 295, 299
- condition codes register, 257
- constructing a truth map, 230
- constructing logic functions using only NAND gates, 238
- constructing minterms from the canonical form, 225
- contention for the bus, 270
- contiguousarray type (Swift), 180
- control bus, 134
- control characters, 96
- control transfer instructions (on the Y86), 292
- control transmissions (USB), 377
- control units, 263
- converting between canonical forms, 229
- copy on write, 117
- counters, 248
- CPI (clocks per instruction), 275
- CPU (central processing unit), 131
 - memory access, 154
- cylinders (on a disk drive), 385

D

- D (data) flip-flop, 246
- dangling pointers, 117
- data bus, 132

- data hazard, 270
- data transfer rates, 357
- dates, 51
 - comparing, 53
- DBCS (double-byte character sets), 100
- debouncing keyboards, 414
- decimal numbering system, 11
- decimal-to-binary conversion, 13
- declaring arrays in memory, 167
- declaring multidimensional arrays, 177
- decoder circuits, 242
- decoding delays, 150
- decoding instruction opcodes, 243
- defragment operation, 408
- delete() memory allocation function, 161, 342
- Delphi strings, 118
- DeMorgan's Theorems, 220
- denormalized operand (floating-point exception), 75
- denormalized values, 71
- descriptor-based strings, 114
- device drivers, 364
- digital audio recorder, 421
- digital design, 217
- Digital Linear Tape (DLT), 392
- digital playback (audio), 420
- digital wavetable synthesis, 420
- direct-mapped caches, 326
- direct memory access (DMA), 354, 355
- direct memory addressing mode, 155, 293
- dirty bits, 331
- disadvantage of the bitmap scheme, 399
- disadvantages of FAT, 402
- disadvantages of zero-terminated strings, 111
- discriminant unions, 187
- disk caches, 387
- disk directory, 398
- disk-drive geometries, 387
- diskless workstations, 381
- dispose() memory allocation function, 161, 342
- distributed shared memory (DSM), 321
- distributive law, 219
- division by zero (floating-point exception), 74
- DLT (Digital Linear Tape), 392
- DMA (direct memory access), 354, 355
- double-byte character sets, 100, 101
- double-precision floating-point format, 69

- double-word storage in byte-addressable memory, 136
- down codes on the keyboard, 415
- down key code, 414
- dual I/O ports, 352
- duality principle, 220
- DVD+RW, 391
- DVD-R, 391
- DVD-RAM, 391
- DVD-ROM, 391
- DVD-RW, 391
- dword, 21
- dynamic memory allocation, 161
- dynamic range in floating-point numbers, 62
- dynamic strings, 117

E

- eager comparison, 66
- EBCDIC (Extended Binary Coded Decimal Interchange Code)
 - character set, 99
- EEPROM (electrically erasable programmable read-only memory), 393
- effective address, 156
- EISA bus, 357
- electrically erasable programmable read-only memory (EEPROM), 393
- encapsulation, 205
- encoding instructions, 285
 - 80x86, 301
 - Y86, 293, 296
- endianness, 143
 - conversion, 144
- end-of-line character, 98
- equivalence function
 - (exclusive-NOR), 222
- error accumulation in floating-point calculations, 63
- exceptions in floating-point arithmetic, 74
- excess-127 exponent, 68
- excess-1,023 exponent, 69
- excess-16,383 exponents, 69, 70
- exclusive-NOR, 222
- exclusive-OR (XOR), 42, 221, 222
- executing instructions out of order, 277
- execution units, 254, 275
- expansion opcodes, 294
- exponents, 62

- Extended Binary Coded Decimal Interchange Code (EBCDIC), 99
- extended-precision floating-point format, 69
- external fragmentation
 - in filesystems, 398
 - in a memory manager, 344

F

- falling edge of a clock, 148
- Fast SCSI, 369
- Fast and Wide SCSI, 369
- FAT (file allocation table), 400, 402
- Fibre Channel, 374
- fields in a record/structure, 181
- file access performance, 407
- file allocation strategies, 399
- file allocation table (FAT), 400, 402
- file fragmentation, 408
- file manager, 396
- file storage (in the memory hierarchy), 321
- filesystems, 396
 - bitmap scheme pros and cons, 399
 - defragmenting, 408
 - directory, 398
 - FAT pros and cons, 402
 - fragmentation, 398
 - free-space bitmaps, 399
 - lists of blocks allocation scheme, 403
 - performance of, 407
 - sequential, 397
 - synchronous I/O, 409
 - three-tiered block scheme, 405
- fine-grained parallelism, 280
- first-fit memory allocation, 343
- first-in, first-out (FIFO) cache
 - replacement policy, 330
- fixed-point representation, 30
- flags register, 257
- flash drive write performance, 395
- flash storage, 393
- flip-flops, 245
- floating-point arithmetic, 61
- floating-point comparisons, 65
- floating-point division, 90
- floating-point exceptions, 74
- floating-point formats, 66
 - double-precision, 69
 - extended-precision, 69
 - quad-precision, 70
 - single-precision, 67

- floating-point operations
 - addition, 75
 - division, 86, 90
 - multiplication, 86
 - subtraction, 75
- floating-point unit (FPU), 30
- floor() function, 72
- floppy disk drives, 382
- floptical drives, 389
- flushing the pipeline, 271
- FM synthesis, 420
- forcing bits to 0, 44
- forcing bits to 1, 44
- formatted text I/O (files), 409
- four-way set associative caches, 328
- FPU (floating-point unit), 30
- fragmentation, 408
- free() memory allocation function, 161, 346
- free-space bitmaps (filesystems), 399
- full adder, 240
- function numbers, Boolean, 222
- functional units, 263
- fusion drive, 396

G

- game controllers, 419
- garbage collection, 117, 161, 345
- General MIDI (GM), 422
- general protection fault, 339
- generics, 213
- geometries of disk drives, 387
- glyphs, 10, 103
- granularity of memory allocation, 347
- grapheme clusters, 103, 105
- great code, 5
- guard bits, 71
- guard digits, 63

H

- half adder, 240
- handshaking, 361
- hard drives, 382
- hardcopy storage (in the memory hierarchy), 322
- Harvard architecture, 272
- heap, 161, 343
- hexadecimal numbering system, 15
- hexadecimal-to-binary conversions, 16
- high-order bit, 21
- high-order byte, 21

- high-speed devices, 357
- HLA (High-Level Assembly) strings, 112
- HLA (High-Level Assembly) unions, 190
- hot-pluggable devices, 374
- hot-swappable devices, 369
- hybrid drives, 396
- HyCode character set, 122
- Hz (hertz), 148

I

- IDE/ATA interface, 372
- identity elements for a Boolean operator, 219
- IEEE floating-point formats, 66
- implementing arrays in Swift, 179
- implementing pointers, 160
- implication (logical function), 222
- indexed addressing mode, 156, 293
- indirect addressing mode, 155, 293
- Industry Standard Architecture (ISA)
 - bus, 357
- inexact result (floating-point exception), 75
- infinity (floating-point), 73
- information hiding (encapsulation), 205
- inheritance, 194, 196
- inhibition (logical function), 222
- initializing arrays in Swift, 168
- input ports, 350
- instruction design goals, 285
- instruction pointer register, 253
- instruction set architecture (ISA), 284
- interfaces (Java), 210
- interleaving sectors on a disk drive, 386
- internal fragmentation, 347, 398
- interrupt service routine (ISR), 364
- interrupt transfers (USB), 378
- interrupts, 363
- invalid operation (floating-point exception), 74
- inverse element (for Boolean operators), 219
- inverted page tables, 335
- inverting bits, 44
- I/O (input/output), 131
 - direct memory access (DMA), 355, 356
 - I/O-mapped I/O, 355
 - memory-mapped I/O, 354–355
 - speed hierarchy, 356
- ISA (Industry Standard Architecture)
 - bus, 357

ISA (instruction set architecture), 284
ISO transfers (USB), 379
ISR (interrupt service routine), 364

J

Java interfaces, 210
Java strings, 114
jnz instruction, 255, 258
joysticks, 419
jump instructions, 299

K

kernel mode (CPU), 346
keyboard
 keybounce, 414
 modifier keys, 415
 scan code, 415

L

L1 cache, 320
L2 cache, 320
L3 cache, 321
latency (of a cache access), 325
Latin-1 character set, 105
least recently used (LRU) cache
 replacement algorithm, 330
least significant bit, 21
left associative operations, 220
legacy peripherals, 413
legacy support, 284
length-prefixed strings, 111
levels
 cache, 320–321
 RAID, 388–389
Linear Tape-Open (LTO), 392
line feed character, 97
list representation of character sets, 120
lists of blocks allocation scheme
 (filesystems), 403
literals, Boolean, 220, 225
little-endian data organization, 142
 issues when using unions, 192
local bus for a CPU, 357
locality of reference, 323
logical AND operation, 42, 218, 222
logical complement operation, 218
logical exclusive-OR operation, 42, 222
logical inhibition, 221
logical NAND operation, 222
logical NOR operation, 222

logical NOT operation, 42, 44, 218, 222
logical OR operation, 42, 218, 222
logical XOR operation, 42
long word, 22
loop instruction, 255, 259
lowercase alphabetic characters, 97
low-order bit, 21
low-order byte, 21
low-speed devices, 356
LPT printer ports, 415
LTO (Linear Tape-Open), 392

M

machine code, 296
machine organization, 3
malloc() function, 161, 346
 string data allocation and, 116
mantissa, 62
mapping method for Boolean function
 simplification, 229
masking, 44
mass storage device filesystems, 396
maximum addressable memory, 133
medium-speed devices, 357
memory
 access, 131, 148
 addressing modes, 154
 allocation, 161
 search algorithms, 343
 best-fit algorithm in a memory
 allocator, 343
 first-fit algorithm in a memory
 allocator, 343
 granularity of memory
 allocation, 347
 leaks, 117
 organization, 131
 protection, 332
 storage of records, 184
 virtual, 332
memory banks, 138
memory cells, 245
memory paging, 332
memory management unit (MMU), 335
memory-mapped files, 335, 410
memory-mapped I/O, 354
mice, 417
microcode, 254
microinstructions, 254
MIDI (Musical Instrument Digital
 Interface) files, 422

- MIMD (multiple instruction, multiple data) execution model, 279
- miniport drivers, 372, 373
- misery approach to comparing floating-point numbers, 66
- modifier keys (keyboard), 415
- mod-reg-r/m byte (addressing mode byte on x86), 303
- modulo-*n* counters, 47
- most significant bit, 21
- mov instruction, 256, 298
- multidimensional arrays, 172, 177–178
- multilevel page tables, 334
- multiple inheritance, 206
- multiple instruction, multiple data (MIMD) execution model, 279
- multiprocessing, 280
- Musical Instrument Digital Interface (MIDI) files, 422

N

- NaN representations, 73
- NAND gate, 238
- NAND operation, 222
- near-line memory storage systems, 322
- near-line storage subsystems, 390
- network storage (in the memory hierarchy), 321
- new() memory allocation function, 161, 342
- nibbles, 20
- Non-Uniform Memory Access (NUMA), 321
- nonvolatile storage, 393
- normal forms (Unicode), 106
- normalization, 70
 - in Unicode, 105, 106
- NOR operation, 222
- NOT instruction (Y86), 298
- NOT operation, 42, 44, 218, 222
- NuBus, 357
- NULL pointer references, 339
- NUMA (Non-Uniform Memory Access), 321
- number of Boolean functions, 221
- numbering systems, 11
- numbers, definition of 10
- numeric digit characters, 97
- numeric overflow (floating-point exception), 75
- numeric representation, 9

- numeric/string conversions, 18
- numeric underflow (floating-point exception), 75
- n*-way set associative cache, 328

O

- octal (base-8) numbering system, 17
- offline storage subsystems, 322
- one-way set associative cache, 326
- online memory subsystems, 322
- opcodes, 254, 285
 - assigning, 290
- operating system file managers, 396
- operation codes. *See* opcodes
- operations involving infinity, 74
- optical drives, 389
- ordinal data types, 169
- OR operation, 42, 218, 222
- OS API calls, 346
- out-of-order execution, 277
- output ports, 350
- overhead of OS API calls, 346

P

- packed data, 51
- packing data, 55
- paging, memory, 332
- parallelism, 260
- parallel ports
 - acknowledge line, 416
 - busy line, 416
 - strobe line, 416
- parallel processing, 279
- Pascal
 - arrays, 169
 - records, 181
 - unions (case-variant records), 187
- PATA interface, 374
- patch-board programming, 253
- PC parallel printer port, 350
- PCI (Peripheral Component Interconnect) bus, 357
- PC peripherals, 413
- peer-to-peer buses, 370
- performance loss due to memory allocation, 348
- performance of OS API calls, 346
- Peripheral Component Interconnect (PCI) bus, 357
- peripheral devices, PC, 413
- pipeline flushing, 271

- pipeline hazards, 273
- pipeline stalls, 271
- pipelining, 267
- platter (hard disk media), 383
- plug-and-play devices, 374
- pointer arithmetic, 162
- pointers, 342
 - address assignment in byte-addressable memory, 162
 - base addresses (of an allocated block), 162
 - types of, 159, 160
- pointing devices, 417
- polled I/O, 363
- polymorphism, 201
- positional notation system, 11
- powerset representation of character sets, 119
- precedence, 220
- prefetch queues, 265
- principle of duality, 220
- private keyword (C++), 205
- processor size, 132
- product of maxterms
 - canonical form, 228
 - representation, 224
- programmed I/O, 355
- programming audio devices, 423
- program status word. *See* flags register
- protected keyword (C++), 205
- protected-mode operation, 364
- protocols (Swift), 210
- pseudo-dynamic strings, 116
- public keyword (C++), 205
- Python strings, 116

Q

- QNaN (quiet not-a-number), 73
- quad-precision floating-point format, 70
- quad word, 22
- quiet not-a-number (QNaN), 73

R

- radix point, 12
- RAID (redundant array of inexpensive disks) systems, 388–389
- RAM disks, 395
- random logic CPU design, 254
- rational representation of fractional values, 35
- read control line, 134

- reading from memory, 136
- read-only port (I/O), 350
- read operations on the bus, 149
- read/write ports, 351
- record base address, 185
- records
 - in C/C++ (structs), 182
 - in HLA, 183
 - in Pascal, 181
 - in Swift (tuples), 183
- reduced instruction set computer (RISC), 260
- redundant array of inexpensive disks (RAID), 388–389
- reel-to-reel drives, 393
- reference counters, 118
 - reference counting for strings, 117
- registers
 - electronic implementation of, 247
 - in the memory hierarchy, 320
 - renaming, 277
- relative-position pointing devices, 417
- replacement policy for caches, 329
- representing arrays in memory, 170
- representing character sets, 119–120
- representing dates, 51
- right associative operations, 220
- RISC (reduced instruction set computer), 260
- rising edge of a clock, 148
- robotic jukebox (optical storage), 390
- rotational latency (of a disk drive), 386
- rounding floating-point results, 71–72
- row-major ordering, 173

S

- sampling (audio), 421
- SATA interface, 374
- saturation, 28
- scalar, Unicode, 102
- scaled-index addressing modes, 157, 306
- scaled-index byte (sib), 307
- scaled numeric formats, 33
- scan code, keyboard, 415
- schematic diagram symbols, 238
- scientific notation, 62
- SCSI (Small Computer System Interface), 367
 - command set, 370
 - miniport drivers, 372
- sectors (on a disk), 383
- semiconductor (RAM) disks, 395

- sequential filesystems, 397
- sequential logic, 245
- serialized operations, 147
- serial ports, 417
- set/reset (SR) flip-flop, 245
- seven-segment decoder, 241
- shift left operation, 48
- shift registers, 247
- shift right operation, 48
- sib (scaled-index byte), 307
- signaling not-a-number (SNaN), 73
- signals (interrupts), 364
- sign bit, 23
- sign contraction, 26
- sign extension, 26
- signed integer values, 23
- significant digits, 62
- simplification of Boolean functions, 229
- single instruction, multiple data (SIMD)
 - execution model, 279
- single instruction, single data (SISD)
 - execution model, 279
- single-precision floating-point
 - format, 67
- size of a processor, 132
- sizeof() function (C/C++), 162
- Small Computer System Interface (SCSI), 367, 370, 372
- SNaN (signaling not-a-number), 73
- solid-state drives (SSDs), 381, 395, 396
- sound cards, 419
- spatial locality of reference,
 - 151, 322, 323
- SSDs (solid-state drives), 381, 395, 396
- stack-pointer register, 342
- stale character data, 117
- starvation (USB), 378
- static strings, 116
- stored program computer, 253
- storing double words in byte-
 - addressable memory, 136
- storing words in byte-addressable
 - memory, 136
- strain gauges (pointing devices), 418
- stralloc() memory allocation
 - function, 117
- streaming data, 391
- streaming tape drives, 393
- strings
 - C#, 115
 - Delphi, 118
 - dynamic, 117

- formats, 110
- Java, 114
- pseudo-dynamic, 116
- Python, 116
- reference counting, 117
- static, 116
- Swift, 115
- zero-terminated, 110
- strobe line (parallel port), 416
- struct assembler directive, 183
- structs in C/C++, 182
- structures, 181–183
- subtracting an integer from a
 - pointer, 164
- subtracting a pointer from a
 - pointer, 164
- sum of minterms, 224, 225, 227
- superscalar operation, 275–277
- surrogate code points in Unicode,
 - 102, 103

Swift

- arrays, 167
- implementation, 179
- initialization, 168
- protocols, 210
- records (tuples), 183
- strings, 115
- unions, 189
- synchronous I/O (filesystems), 409
- system buses, 132, 357
- system clock, 147
 - period, 148

T

- tally/slash numeric representation, 11
- tape drives, 392
- tbyte, 22
- templates, 213
- temporal locality of reference,
 - 151, 322, 323
- terms (Boolean), 223, 224
- testing bits for 0 or 1, 45
- theorems of Boolean algebra, 219
- thrashing (cache), 331
- three-level page tables, 335
- three-tiered block scheme
 - (filesystems), 405
- timeouts on peripheral devices, 362
- TLB (translation lookaside buffer), 334
- touch screens, 418
- trackpads, 417
- trackpoints, 418

- tracks (on a disk), 383
 - track-to-track seek time, 384
- translation lookaside buffer (TLB), 334
- truncating floating-point numbers, 71
- truth tables, 43, 220
- tuples (records) in Swift, 183
- two's complement numbering system, 23
- two-level caching system, 153
- two-way set associative caches, 328

U

- unconditional jumps (Y86), 299
- Unicode
 - BMP (Basic Multilingual Plane), 102
 - canonical equivalence, 105
 - character names, 103
 - code planes, 102
 - code points, 101
 - combining characters, 108, 109
 - encodings, 107, 108
 - UTF-8, 107
 - UTF-16, 107
 - UTF-32, 107
 - multilingual planes, 102
 - normal forms, 106
 - scalar, 102
 - surrogate code points, 102
 - Unicode character set, 101
- uninitialized data sections in
 - memory, 341
- unions
 - in C/C++, 187
 - endian issues, 192
 - in HLA, 190
 - memory storage, 190
- unique Boolean functions, 221
- universal Boolean function (NAND gate), 238
- Universal Serial Bus. *See* USB
- unpacking data, 55
- unsigned integer values, 23
- up codes on the keyboard, 415
- uppercase alphabetic characters, 97
- USB, 374–375
 - bulk transmissions, 377, 378
 - bus enumeration, 380
 - client drivers, 380
 - control transmissions, 377
 - host controller stack, 376
 - interrupt transmissions, 377, 378
 - isochronous transmissions, 377, 379

- starvation, 378

- USB-c, 379

- user mode (CPU), 346

- UTF-8 encoding (Unicode), 107

- UTF-16 encoding (Unicode), 107

- UTF-32 encoding (Unicode), 107

V

- variable-length instructions, 289
- very long instruction word (VLIW)
 - architectures, 278, 279
- virtual memory, 321, 332
- virtual method tables (VMTs), 194
- virtual sound canvas, 423
- von Neumann architecture, 131

W

- wait states, 150
- wavetable synthesis, 420
- Wide SCSI, 369
- words, 21
 - accessing in byte addressable
 - memory, 136
 - stored at odd addresses, 139
- working sets, 334
- WORM (write-once, read-many), 391
- write control line, 134
- write lifetime (flash device), 394
- write-once, read-many (WORM), 391
- write-only port (I/O), 350
- write operations on the bus, 149
- write performance of flash drives, 395
- write-through cache write policy, 330
- writing to memory, 135

X

- XOR (exclusive-OR) operation, 42, 222

Y

- Y86 hypothetical processor, 291

- Y86 instructions, 291

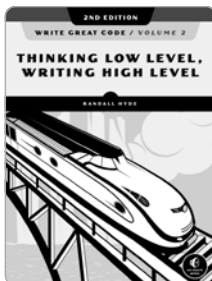
Z

- zero extension, 26
- zero flag, 255
- zero-terminated strings, 110
 - disadvantages, 111

RESOURCES

Visit https://nostarch.com/writegreatcode1_2e for resources, errata, and more information.

More no-nonsense books from  **NO STARCH PRESS**



WRITE GREAT CODE, VOLUME 2, 2ND EDITION

Thinking Low-Level, Writing High-Level

by RANDALL HYDE

JULY 2020, 656 PP., \$49.95

ISBN: 978-1-71850-038-9



WRITE GREAT CODE, VOLUME 3 Engineering Software

by RANDALL HYDE

JULY 2020, 360 PP., \$49.95

ISBN 978-1-59327-979-0



EFFECTIVE C

An Introduction to Professional C Programming

by ROBERT C. SEACORD

JULY 2020, 272 PP., \$59.95

ISBN 978-1-71850-104-1



THE RUST PROGRAMMING LANGUAGE

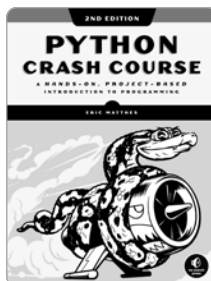
(Covers Rust 2018)

by STEVE KLABNIK AND

CAROL NICHOLS

AUGUST 2019, 560 PP., \$39.95

ISBN 978-1-71850-044-0



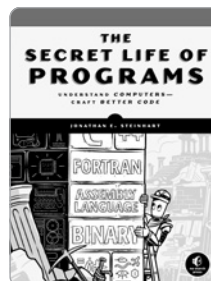
PYTHON CRASH COURSE, 2ND EDITION

A Hands-On, Project-Based Introduction to Programming

by ERIC MATTHESS

MAY 2019, 544 PP., \$39.95

ISBN 978-1-59327-928-8



THE SECRET LIFE OF PROGRAMS

Understand Computers—Craft Better Code

by JONATHAN E. STEINHART

AUGUST 2019, 504 PP., \$44.95

ISBN 978-1-59327-970-7

PHONE:

1.800.420.7240 OR

1.415.863.9900

EMAIL:

SALES@NOSTARCH.COM

WEB:

WWW.NOSTARCH.COM



The Electronic Frontier Foundation (EFF) is the leading organization defending civil liberties in the digital world. We defend free speech on the Internet, fight illegal surveillance, promote the rights of innovators to develop new digital technologies, and work to ensure that the rights and freedoms we enjoy are enhanced — rather than eroded — as our use of technology grows.



EFF.ORG

ELECTRONIC FRONTIER FOUNDATION

Protecting Rights and Promoting Freedom on the Electronic Frontier

MACHINE ARCHITECTURE FOR MERE MORTALS



This first volume of Randall Hyde's classic *Write Great Code* series dives into machine organization without the extra overhead of learning assembly language programming. Written for high-level language programmers, *Understanding the Machine* fills in the low-level details of machine organization that are often left out of computer science and engineering courses. You'll learn:

- How the machine represents numbers, strings, and high-level data structures, so you'll know the inherent cost of using them
- How to organize your data, so the machine can access it efficiently
- How the CPU operates, so you can write code that works the way the machine does
- How I/O devices operate, so you can maximize your application's performance when accessing those devices
- How to best use the memory hierarchy to produce the fastest possible programs

Great code is efficient code. But before you can write truly efficient code, you must understand how computer

NEW COVERAGE OF:

- Programming languages like Swift and Java
- Code generation on modern 64-bit CPUs
- ARM processors on mobile phones and tablets
- Newer peripheral devices
- Larger memory systems and large-scale SSDs

systems execute programs and how abstractions in programming languages map to the machine's low level hardware. After all, compilers don't write the best machine code; programmers do. This book gives you the foundation upon which all great software is built.

ABOUT THE AUTHOR

Randall Hyde is the author of *The Art of Assembly Language* and the three volume *Write Great Code* series (all No Starch Press). He is also the co-author of *The Waite Group's MASM 6.0 Bible*. He has written for *Dr. Dobbs' Journal* and *Byte*, and professional and academic journals.



THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

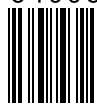
\$49.95 (\$65.95 CDN)

ISBN 978-1-7185-0036-5

5 4 9 9 5



9 781718 500365



SHELF IN:
COMPUTERS/PROGRAMMING